Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes

Tuojian Lyu

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 28.06.2020

Supervisor

Dr. Sierla Seppo

Advisor

Miro Eklund

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Tuojian Lyu

Title Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes

Degree programme ICT Innovation

Major Cloud Computing and Services **Code of major** SCI3081

Supervisor Dr. Sierla Seppo

Advisor Miro Eklund

Date 28.06.2020 **Number of pages** 59+16 **Language** English

Abstract

The modern industrial systems are large and complex so that a new simulation method, cooperative simulation or co-simulation, is used to simulate sub-models of a whole system. A large and complex system will be divided into several smaller subsystems, and these smaller systems will be modeled and simulated by multiple cooperative simulators. This simulation method enables the simulation process to be efficient and can provide many advantages, such as viewing results in real-time, consuming resources efficiently, and providing more accurate results than simulating the whole large and complex system. Besides the co-simulation method, this thesis also introduces the Docker container technology, a container virtualization tool used to build and pull images, run containers, and orchestrate containers. Another container orchestrating tool, Kubernetes, is also used in the experiment for managing pods and containers. This thesis discusses the possibility of containerizing simulation software in Docker, and uses Docker swarm and Kubernetes to orchestrate cooperative simulation containers. A co-simulation platform is created in a Docker swarm cluster and Kubernetes cluster, where multiple simulation containers are running cooperatively by receiving commands from the co-simulation platform. The experiment results prove that the co-simulation platform is working as expected, and that multiple cooperative simulation containers have better performance than running a standalone complex simulation process.

Keywords Simulation technology, Container virtualization, Docker, Kubernetes, Co-simulation

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Tuojian Lyu

Työn nimi Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes

Koulutusohjelma ICT Innovation

Pääaine Cloud Computing and Services **Pääaineen koodi** SCI3081

Työn valvoja Dr. Sierla Seppo

Työn ohjaaja Miro Eklund

Päivämäärä 28.06.2020 **Sivumäärä** 59+16 **Kieli** Englanti

Tiivistelmä

Nykyiset teollisuusjärjestelmät ovat suuria ja monimutkaisia, joten uutta simulointimenetelmää, yhteistoiminnallista simulaatiota tai niin simulointia, käytetään koko järjestelmän alamallien simulointiin. Suuri ja monimutkainen järjestelmä jaetaan useisiin pienempiin osajärjestelmiin, ja nämä pienemmät järjestelmät mallinnetaan ja simuloidaan useilla yhteistoiminnallisilla simulaattoreilla. Tämä simulaatiomenetelmä mahdollistaa simulointiprosessin tehokkaan ja voi tarjota monia etuja, kuten tulosten tarkasteleminen reaaliajassa, resurssien tehokas kulutus ja tarkempien tulosten tarjoaminen kuin koko suuren ja monimutkaisen järjestelmän simulointi. Yhteissimulaatiomenetelmän lisäksi tämä opinnäytetyö esittelee myös Docker-säilytysteknologian, konttien virtualisointityökalun, jota käytetään kuvien rakentamiseen ja vetämiseen, säilöiden ajamiseen ja säiliöiden järjestämiseen. Toista kontinorkerointityökalua, Kubernetesia, käytetään myös kokeessa palkojen ja astioiden hallintaan. Tässä opinnäytetyössä tarkastellaan simulaatio-ohjelmistojen konttimahdollisuuksia Dockerissa ja Docker-parven ja Kubernetesin avulla organisoidaan yhteistyöhakuisia simulaatiosäiliöitä. Docker-parvi- ja Kubernetes-klusteriin luodaan yhteissimulaatiofoorumi, jossa useita simulaatiosäiliöitä toimii yhteistyössä vastaanottamalla komentoja simulointiympäristöltä. Yhteissimulaatiofoorumi toimii odotetusti, ja koetulokset osoittavat, että useiden yhteistyöhakuisten simulaatiosäiliöiden suorituskyky on parempi kuin erillisen monimutkaisen simulaatioprosessin suorittamisen.

Avainsanat Simulator, Container Virtualization, Docker, Kubernetes, Co-simulation

Aalto University
School of Science

Aalto-universitetet, PB 11000, 00076 AALTO
www.aalto.fi
Sammandrag av diplomarbetet

Författare Tuojian Lyu

Titel Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes

Utbildningsprogram ICT Innovation

Huvudämne Cloud Computing and Services **Huvudämnets kod** SCI3081

Övervakare Dr. Sierla Seppo

Handledare Miro Eklund

Datum 28.06.2020 **Sidantal** 59+16 **Språk** Engelska

Sammandrag

Sammandrag på svenska. De nuvarande industrisystemen är stora och komplexa så att en ny simuleringsmetod, kooperativ simulering eller co-simulering, används för att simulera delmodeller av ett helt system. Ett stort och komplext system kommer att delas upp i flera mindre delsystem, och dessa mindre system kommer att modelleras och simuleras av flera kooperativa simulatorer. Denna simuleringsmetod möjliggör effektiv simuleringsprocess och kan ge många fördelar, såsom att se resultat i realtid, konsumera resurser effektivt och ge mer exakta resultat än att simulera hela det stora och komplexa systemet. Förutom samsimuleringsmetoden introducerar denna avhandling också Docker-containerteknologin, ett containervirtualiseringsverktyg som används för att bygga och dra bilder, köra containrar och orkestrera containrar. Ett annat containerorkestreringsverktyg, Kubernetes, används också i experimentet för att hantera fröskidor och containrar. Den här avhandlingen diskuterar möjligheten att containerisera simuleringsprogramvara i Docker och använder Docker swarm och Kubernetes för att orkestrera kooperativa simuleringsbehållare. En samsimuleringsplattform skapas i Docker-svärmklustret och Kubernetes-klustret, där flera simuleringsbehållare körs samarbete genom att ta emot kommandon från samsimuleringsplattformen. Samsimuleringsplattformen fungerar som förväntat, och experimentresultaten bevisar att flera kooperativa simuleringsbehållare har bättre prestanda än att köra en fristående komplex simuleringsprocess.

Nyckelord Simulator, Container Virtualization, Docker, Kubernetes, Co-simulation

Contents

Abbreviations

VR	Virtual reality
AR	Augmented reality
IoT	Internet of things
GCP	Google Cloud Platform
XML	Extensible markup language
UI	User interface
Hil	Hardware in the Loop
SITL	System in the Loop
DLL	Dynamic link library
PLC	Programmable logic controller
SOA	Service-Oriented architecture
SOAP	Simple object access protocol
COM	Component object model
DCOM	Distributed component object model
OS	Operating system
YAML	YAML ain't markup language
IaaS	Infrastructure as a service
PaaS	Platform as a service
SaaS	Software as a service
CoW	Copy on write
CLI	Command-line interface
REST	Representational state transfer
API	Application program interface
SCL	Structured control language
URL	Uniform resource locator
OPC UA	OPC unified architecture
OPC DA	OPC data access
OPC HDA	OPC historical data access

1 Introduction

Manufacturing enterprises are trying to transfer their production mode from digital to intelligent mode where many smart technologies are used, e.g., VR/AR and IoT. This trend is because many technologies are developed quickly, such as information technology, big data and cloud computing. Transferring from digital to intelligent production mode makes the traditional manufacturing industries diminished, and the intelligent production mode gradually becomes the mainstream.

An increasing number of countries are trying to benefit from this trend and even want to play a critical role in this change. EU proposed the 2020 strategy [1] to take charge of long-term challenges – globalization, stressing on resources and aging–intensify while Industry 4.0 [2] enhanced Germany's role in the industry. Meanwhile, China introduced the China Manufacturing 2025 [3] to guarantee its leading role in the manufacturing area. America, in October 2018, released the Strategy for America Leadership in Advanced Manufacturing [4], to ensure security and economic prosperity its leadership in advanced manufacturing across all industrial sectors. There are more and more suggestions, propositions, and strategies introduced by many other countries to transform the intelligent manufacturing model [5].

Industry 4.0 enhances the reliability of the connection between the virtual and physical world. Because of that, smart systems now can be linked through advanced technologies like IoTs. Multiple intelligent systems can be combined and operated together. Thus cyber-physical systems can utilize them to achieve bigger goals and better performance.

The utilization of multiple smart systems brings the new need of simulation technology. In a simulation system, it evaluates the overall performance of all different intelligent systems that run as components independently. Using a simulation system to simulate involved subsystems could ensure that all the elements work both independently and cooperatively in real-time. For instance, it is often used in processes or intelligent systems before cyber-physical systems can use them in real scenarios.

Simulation technology is an essential component integrated into intelligent systems to realize functionalities and features of smart systems as well. Typical functions and features of intelligent systems in Industry 4.0 include self-optimization, self-diagnosis and self-networking. Smart systems are widely used in scenarios and fields for these characteristics. The development and verification of these systems require simulation support, thus leading to the demand and interest of simulation technology. Now, many simulation software and platforms are used to train operators, optimize user interfaces, and secure the manufacturing processes. To utilize simulation in smart systems appropriately, structures and methods need to be designed and implemented carefully to ensure that multiple simulators run cooperatively and synchronously [6].

Collaborative Digital Twin is an excellent option to cooperate with those complicated and interactive smart systems. Collaborative digital twin refers to a digital twin that can operate multilaterally in the cloud. It enables different actors, such as equipment providers and system owners, to do the test in the cloud with their own equipment or customized options. For example, an equipment provider can test a specific valve for the existing system to check if this new valve is a better option than the existing

one. There are many smart systems or collaborative digital twins that have to utilize the simulation technology to achieve the optimization objective described above. To achieve this objective, three critical issues to be considered are listed below.

- Which simulation technology or simulator software should be used.
- How to design the co-simulation platform.
- How to make multiple simulators communicate synchronously and efficiently.

These three issues are the fundamental problems of running multiple cooperative simulators, and the details of the goals and scopes of this thesis will be introduced in section 1.2. This thesis presents how to deploy multiple cooperated simulators for collaborative digital twin using Docker and Kubernetes technologies based on the need for continuous and cooperative simulation in the cloud. Scalability and stability are the two primary measurements for the work of this thesis. These two measurements will be discussed in detail in chapter 3. The motivation for this thesis is introduced in section 1.1. Section 1.2 presents the goals and scopes of the thesis. Finally, the overall structure of this thesis can be found in section 1.3.

1.1 Motivation

The first motivation of this thesis is to evaluate and compare the performance between containerized simulators and standalone simulators according to scalability and flexibility. Docker and Kubernetes technologies are used in this paper to containerize simulations software, which, in this thesis, refers to Apros as an example. The second motivation is to investigate the combination of advantages of cloud technologies and container technologies by creating the Docker and Kubernetes container environment on the Google Cloud Platform (GCP).
Cloud computing brings advantages to facilitate simulation technologies. For example, computing and storage resources are pay-per-use [7] and almost unlimited, with which companies can do multiple simultaneous simulation processes. It also stimulates a business model of providing such simulation service as Service-as-a-Service that brings a win-win scenario to both providers and users. While companies who provide simulation services could utilize the computing and storage resources on the cloud to fulfill compelling simulation services, users who use these services do not need to manage and organize complex clusters for continuous real-time operations. Based on the two motivations above, this paper investigates the implementation and performance of cooperative containerized simulators. The implementation divides a complex simulation model into smaller sub-models so that each of the sub-model utilizes the computation and memory resources separately and efficiently. The co-simulation technology provides advantages, including scalability, security, and collaboration, using cloud-related technologies to simulate sub-models cooperatively on the cloud platform. To achieve these advantages, it is worth investigating and designing a co-simulation platform where sub-models can be simulated by containerized simulation software on the cloud automatically, and these simulation containers can be

organized and scaled by Kubernetes efficiently.
In conclusion, this thesis's motivation is to implement the method of deploying simulators by using container and cloud technologies, and secondly, to evaluate and compare the performance of this containerized cloud-based simulators with standalone simulator processes.

1.2 Goals and Scopes of the Thesis

This thesis aims to test if the Docker container technology would be beneficial for co-simulation platform development. Specifically, this thesis investigates whether the containerized simulators can perform better than standalone simulator processes according to different aspects, such as scalability and flexibility.
The primary goal is to design and test a co-simulation platform where simulation software processes can be containerized by Docker technology and managed by Kubernetes technology. To achieve this goal, the co-simulation platform is implemented and evaluated using different real testing cases. The testing process is that the simulation software will be containerized and run on the GCP, and the OPC UA will be used to enable these two simulation containers to communicate. Finally, the performance of containerized cloud-based simulators and standalone simulator processes can be compared with the same testing cases according to different aspects, such as scalability and flexibility.
Therefore, the main objective of this thesis is divided into several sub-tasks:

- **Making the Dockerfile of the Simulation Software:** In order to deploy a specific simulator container, it is necessary to make a customized Dockerfile for that simulation software without UI so that the simulator image can be built via that Dockerfile.

- **Implementing OPC UA Configuration XML Files:** Before moving to the part of running simulators in containers, OPC UA interfaces of two simulation software instances are made and tested on-premise. According to different purposes, the OPC UA interfaces configuration files should be designed and tested carefully on-premise, because it is challenging to troubleshoot inside containers without the simulation software UI. Moreover, different communication models, e.g., mutual subscriptions or single subscription/write, should be considered and tested.

- **Designing Docker Swarm and Kubernetes Orchestration:** Docker swarm and Kubernetes are the two main orchestrations for containers. Both of these two technologies enable users to customize their cluster for running Docker containers, which means Docker swarm and Kubernetes are alternatives with each other, to some extent. Therefore, firstly, it is essential to design the basic orchestration of Docker swarm and then Kubernetes for further testing. The task of this step is to decide some basic configurations for a cluster, e.g., the operating systems of node servers, the number of node servers, and the minimum CPUs and memory.

- **Creating the Docker Swarm and Kubernetes Cluster on the GCP:** Based on the configuration of orchestration of Docker swarm and Kubernetes, the implementation detail of creating Docker swarm and Kubernetes on GCP will be introduced, such as networking configuration, set-up scripts, and testing cases. These two clusters will be created as the experiment platform for running multiple cooperative simulator containers in the GCP.

- **Implementing Kubernetes Java Client:** To deploy simulator containers in Kubernetes automatically, the Kubernetes Java client needs to be designed and implemented. The Kubernetes Java client is the client that utilizes Java client libraries of Kubernetes API so that kubectl commands can be used in the form of Java code and, importantly, deploying simulation software can be automated.

- **Generating the OPC UA XML Files Automatically via the Java Application:** After testing the OPC UA configuration XML files generated automatically, a Java application should be designed and implemented, which can be automatically used in the co-simulation platform to generate the OPC UA configuration files automatically. Correctly, the co-simulation platform will receive the simulation details, e.g., the number of simulations, simulating components, and simulation items, from end-users. And then, the co-simulation platform can call this component to generate the necessary OPC UA configuration XML files, which will be used by simulators for the OPC UA interface.

- **Testing Simulator Containers:** This step is to test the simulator containers in the Docker swarm and Kubernetes environment. Specifically, the OPC UA interfaces will be tested to verify that the OPC UA related configuration files are correct, and the OPC UA interfaces are working well between two simulator containers. The thesis's objective is to implement a method enabling multiple containerized simulators to cooperate and communicate via OPC UA interfaces.

- **Evaluating the Performance of Simulator Containers:** After testing the simultaneous orchestration, the overall performance should be evaluated, in which two simulators communicate via OPC UA interfaces for 500 testing signals and two testing modes. This step's primary motivation is to confirm if the Kubernetes-based simulators can perform better than the standalone simulators.

1.3 Structure of Thesis

The structure of this thesis is as follows. Chapter 2 introduces the related work and the background of simulation technology, OPC interfaces, the Docker, and Kubernetes technologies. Chapter 3 presents the research approach used in this thesis, including the experiment design, experiment description, and the testing tools used in the experiment. Chapter 4 presents methods to implement all necessary components of the co-simulation platform, including the Docker and Kubernetes clusters, Dockerfile

of Apros, and the Kubernetes Java client. Chapter 5 describes different measurements for testing this co-simulation platform and analyzes the collected testing results. Moreover, the performance of the co-simulation platform, Kubernetes Java client, and other components implemented by this thesis is also discussed in chapter 5. The last chapter 6 gives the conclusion about the contribution of this thesis.

2 Background

In this chapter, section 2.1 introduces the related work. Section 2.2 presents the general definition of simulation technology. Section 2.2.1 is the introduction of Apros that is the simulation software used in this thesis experiment. Section 2.3 introduces the OPC specifications. Container technology and two types of container orchestration mechanisms, including the Docker swarm and the Kubernetes, are presented in section 2.4 and section 2.5.

2.1 Related Work

The previous work [8] designed and implemented the core functionalities of a synchronized cooperative simulation platform that is effective and scalable. The most critical implementation is synchronizing OPC UA communication between two simulators. Thus the co-simulation platform can support multiple simulators to cooperate via the synchronized OPC UA interfaces. The evaluation is that both the OPC UA configuration and the synchronization mechanism can achieve expected performance for building the co-simulation platform. However, there are also drawbacks. For example, the Apros used is not the latest version (Apros 6), and each simulator is restricted by the computing resource in the local machine.
Based on the previous work on the synchronization and OPC UA interface configuration for Apros simulators, the container virtualization is added in the experiment of this thesis to provide a more capable and scalable co-simulation platform on the cloud. Specifically, Docker and Kubernetes technologies are used to containerize and manage the Apros simulators on the cloud while multiple containerized simulators can also cooperate via the OPC UA interfaces. Moreover, the evaluation of this container-based cooperative simulators via OPC UA is presented according to the experiment results in chapter 5.

2.2 Simulation Technology

The definition of simulation or modeling is that simulation and modeling are used to simulate or model the real or imagined systems or processes so that people can collect the results and predict or optimize their systems and processes. When referring to the simulation or modeling, it means the computing simulation instead of physical simulation, for example, using stones or wood pieces to represent military units [9]. One definition of computer simulation is that it is a computer program that can analyze and predict a mathematical model's results. When referring to the computer simulation, several steps need to be considered, such as analyzing the simulation or modeling results, how to create a model working on computers, and how to customize the specific simulation model. Another definition of a simulation is that the simulation is the execution of a model, which is a computer program providing information about the simulated system [10]. The question of how to have a better performance with the lowest cost is the main objective for computer system users, administrators and designers. Modeling and computer simulations help users achieve this objective in

the real world. Besides utilizing modeling and simulation to understand and optimize systems' performance, simulation ensures the correctness of the modeling systems. Today's hardware and software should be tested with modeling and simulation before going to be published to market. Moreover, simulation can reduce the cost of repairing crashes or fixing bugs, because the products will be modeled and simulated according to many different aspects to ensure quality and stability. Another scenario where the simulation will be applied is that computer simulation helps develop virtual environments, e.g., pilot training [11] and system operator training.

Dynamic modeling and simulation are fundamental technologies in the industry to check the changes in real-time. The compelling visual modeling and simulation help engineers and developers to predict the results and quickly make decisions in real-time. The system simulation means to emulate the process of real systems, e.g., flight systems, bank systems, and assembly lines. The system simulation technology helps to provide useful and accurate analyzed results of the simulated systems instead of building complex and expensive systems for the experiment.

Since the first modern mathematical simulation obtained huge interests and influence, the simulation and modeling technology have been established and optimized very well. The modern industry simulation and modeling technologies are more and more popular and used in a wide variety of areas, such as food production, nuclear industry, and paper production. Utilizing simulation and modeling technologies can provide many advantages, including reducing developing time and cost, increasing security and safety, and optimizing the existing manufacturing process.

It is necessary and essential to utilize computer simulation technology to simulate dynamic systems. Computer simulation becomes popular because it helps to shorten the development cycles and reduce testing and prototyping costs.

After decades of computer simulation development, the systems in the real-time environment can be simulated, including Hardware-in-the-Loop (Hil) and System-in-the-Loop (SITL) setups. Based on the improvement of modern computer simulation, current simulation technologies are integrated with physical components. Moreover, these simulation processes are mostly related to the simulation of multi-physical systems, which are usually complicated and have many subsystems.

Because the systems become more complex and challenging to be simulated by a single simulation, the problem for modern simulation technology is how to use multiple simulation processes to simulate different subsystems. For example, multiple simulators simulate multiple subsystems of a vehicle according to different testing situations [12].

This problem can be resolved by utilizing a co-simulation mechanism [13], where many simulators simulate subsystems, and each simulator is responsible for a subsystem. In this case, the co-simulation platform must be designed and implemented based on the cooperation of the simulators for each subsystem. Moreover, the communication mechanism among different simulators should also be configured and enabled. One of the examples is that the OPC UA interfaces are enabled for Apros simulators for testing cases in this thesis's experiment.

2.2.1 Introduction of Apros

Apros is a simulation tool used in industrial processes for full-scale modeling and dynamic simulation [14]. There is a wide variety of scenarios where Apros is used as the primary simulation tool. For example, these scenarios are automation suppliers, paper mills, and power plants. The reason why the Apros is widely used in these scenarios is that most of these scenarios require the full-scale modeling and dynamic simulation. Apros can fulfill these requirements and provide many features for these scenarios [8]. Furthermore, Apros is not only a simulation tool used in industry for simulating the systems or processes but also an education software used in many universities around the world [15]. In Finland, many universities are researching on utilizing Apros and educating students simulation technologies with real Apros cases. Apros has several ways of communicating with external applications. The classic OPC DA is one of these methods. The Apros frontend implements the Adda interface. Adda is a proprietary interface with data access and simulation control functionalities. OPCDAKit is a dynamic-link library (DLL) in Apros, which maps the Adda interface to the OPC DA interface. In addition to the OPC DA interface, OPCDAKit implements the Simulation Control (SC) interface. The Simulation Control interface is used alongside with OPC DA to control the simulation [8].
To model the full-scale and simulate the dynamics of a significant number of processes, Apros provides many features, including friendly user interface design, efficient solution algorithms, and many plug-ins. These features enable Apros to be the main option because there is a need to simulate and model complex systems and processes. Because the model libraries were already validated against the real data from the physical process test, Apros also can be utilized to model automation and electrical systems in detail. Each model that can be found in Apros is validated and one-to-one analogous according to one specific device. The main scenarios using Apros are shown as follows:

- Development of control strategies
- Analysis of the system operation
- Verification of design
- Testing of control system
- Training of operators
- Development of operational practices and the control room

To gain the ultimate benefits from the simulation of processes and systems, full-scale modeling, and dynamic simulation should follow the engineering workflow of the design of the plant. The operation and maintenance should also follow the results of the dynamic simulation. The complexity of processes or systems can vary from very simple to ultra-complex like nuclear plants, which means it is sometimes impossible to execute such simulations manually. Alternatively, the semantics-based modeling interface integrated into Apros helps make the simulation of complex processes or

systems easier and more efficient. Moreover, the simulation results can be viewed dynamically in real-time so that the development teams or operators can quickly fix design bugs or collect the experiment results.

2.3 OPC Interfaces

In this section, the detail of the OPC will be introduced. OPC is the standard to access different physical devices in the control and automation systems so that data can be transmitted between devices and systems [16]. This is also why the OPC becomes more and more popular in the industry for data transmission. Section 2.2.1 introduces classic OPC. Section 2.2.2 introduces the most critical and advanced OPC tool that is OPC UA.

2.3.1 Classic OPC

Three major OPC standards were introduced and developed for the increasing number of requirements within industrial areas. These three OPC standards are Data Access (DA), Alarm Events (AE), and Historical Data Access (HDA). OPC DA introduces a way to access current process data, AE adds the interface for information of events or issues, and HDA introduces the way of utilizing functions to access achieved data. All the OPC interfaces mentioned above can navigate the address space and give all the related information about the target item.
The OPC Classic specifications have been developed based on Microsoft Windows technology, which is explicitly utilizing COM/DCOM (Distributed Component Object Model) to provide communication interfaces for software components based on distributed server-client network architecture. These three OPC specifications transmit and display process data, alarmsevents, and historical data among multiple connected devices. In 2010, the new specification, the OPC .NET 4.0, was used to enhance the Classic OPC to provide better performance with new and advanced technologies for end-users [17].
One advantage of OPC is the quality of data, which is also the most critical aspect of transmitting data among different components and devices. OPC provides and transmits data in real-time, even if the current devices are temporarily interrupted. This kind of interruption issue of devices can be resolved by providing timestamps for the transmitted data and provide the quality information according to different situations: accurate (right), not available (bad), and unknown (uncertain) [18]. Figure 1 introduces the classical use cases of OPC specifications.

2.3.2 OPC Data Access

Variables from current process data can be read, written, and monitored via OPC DA interfaces. The OPC DA aims to transmit the data from PLCs, DCSs, and other control devices to the frontend, e.g., HMIs, for display. OPC DA is one of the most critical specifications among all Classic OPC specifications because many products worldwide are using this OPC specification.
The variables which are waiting to be read, written, and monitored should be explicitly

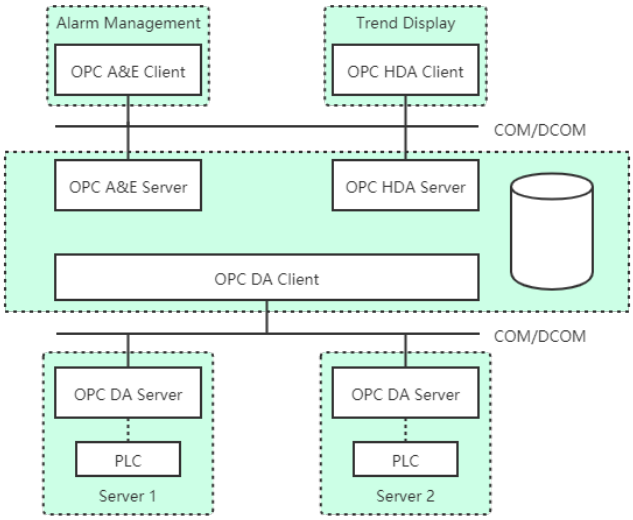

Figure 1: The use cases of OPC specifications.

selected because only the selected data can be shown and operated via OPC DA. Moreover, the OPCServer object should also be created for the OPC client to connect to the server so that the selected variables can be transmitted between the OPC DA client and server. Different items and properties of items can be viewed under the address space hierarchy or through the interfaces provided by the OPC server object [18]. Figure 2 presents the use case of OPC DA and OPC HDA.

2.3.3 OPC Alarm Events

The events and alarms data can be received via the OPC AE interface. There are two types of notifications in the OPC AE. The first type of notification is the alarm, which is the information about the change of the current unexpected or risk condition where some components or processes may be malfunctioning. The water tank can be the example of the alarms generated via OPC AE when the water level reaches the maximum water level limitation. The second type of notification is the event used to notify users of some monitored events. OPC AE thus provides a flexible interface for transmitting process alarms and events from different event sources.
Specifically, if users want to view these alarms and events, they have to connect the OPC AE client to the OPC AE server to subscribe to monitoring items and receive all triggered notifications. Moreover, there will be many notifications generated among all components and processes, but end-users can also reduce the number of notifications by using filter criteria.
When the OPC client connects to the server, the OPCEventServer object will be created in the OPC AE server. And then, the OPCEventSubscribing object will be generated and used to receive the generated notifications.
The OPC AE specification does not require specific information like data type and namespaces. All the events generated from all processes and subsystems will be

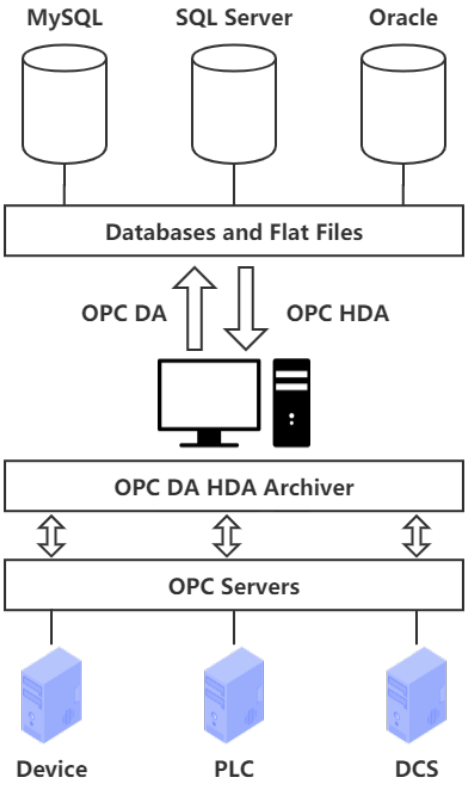

Figure 2: The use case of OPC DA and OPC HDA.

transmitted to OPC AE client, and some filter options, e.g., event types, timestamps and source devices, can be chosen to filter the data according to different purposes.

2.3.4 OPC Data Historical Access

In contrast to the OPC Data Access, which can provide data changes in real-time from processes and subsystems, OPC Historical Data specification provides the interface where the historical stored data can be retrieved later. Similar to the OPC AE, two objects will be created in the server when the OPC HDA connects to the server. The first object is the OPCHDAServer providing all the interfaces used to read and update the historical data. The second object is the OPCHDABrower, which is used to define the browsing address space in the HDA server.
The most crucial function of OPC HDA is to read the data of variables. However, three different mechanisms can be used when reading data of variables. The first mechanism is that one or more variables can be defined according to the range of timestamps. Using this reading mechanism, all the data of chosen variables from that specific time range will be received. The second mechanism is similar to the first one that multiple variables can be chosen, but the timestamps have to be specified instead of the time range. Finally, the third mechanism is used in the scenario where aggregate values are requested according to one or more variables opted for the

specific timestamp. Moreover, the OPC HDA can not only be utilized to read the historical data from databases but also can insert, modify, and delete existing data in the databases.

2.3.5 OPC Unified Architecture

Because of the complexity of Industry 4.0, it is impossible to design and implement a single standard that can be used around the world in the industry area. Many components constitute the modern digital industry with a lot of specific standards that can not be easily replaced by a single standard. Thus the goal of the modern industry is to implement the standard which integrates all necessary standards, e.g., systems of systems. Based on the need to integrate existing OPC specifications, the OPC UA [19] was implemented to combine Classical OPC specifications. The OPC UA provides two significant advantages, which are flexibility and variability for the communication. Specifically, the OPC UA can be used to create communication channels in a wide variety of areas, from simple data to very complex systems.
OPC UA is a vendor-independent communication specification used in industrial automation applications. Moreover, the OPC UA is also the platform-independent communication specification with integrated security mechanisms [20].
The OPC Foundation implements the OPC UA as a new communication specification for reliable and secure transmitting of the data in real-time processes. One unique advantage of OPC UA is that any authorized person can access any permitted data easily. It is easy to transmit and monitor the data under authorization. The OPC UA interfaces can be easily integrated into many existing applications or processes on many platforms. The OPC UA is powerful than any other OPC specification because OPC UA integrates many advantages of other OPC specifications such as scalability, security, and platform-independence. Therefore, the OPC UA plays a critical role in the modern digital industry, and the role is to create the communication bridge between the enterprise level and embedded automated components level [21].

2.3.6 Advantages and Disadvantages

Standard web technologies are used by OPC UA to improve security and make it platform-independent based on classic OPC's functionalities. Many challenges or difficulties were resolved due to the improvements made by OPC UA. Moreover, because of these improvements of OPC UA, this open industry standard becomes more and more popular in the industrial automation and areas where the open and secure communication between devices is required.
The primary motivation for implementing the OPC UA is to utilize the cross-platform, business-optimized Service-Oriented Architecture (SOA) to improve the Classic OPC functionalities. There are two protocols enabled in OPC UA, which are a binary protocol and a web service protocol (SOAP). The binary protocol is used to employ minimal resources for allowing secure enablement through a firewall. Furthermore, the SOAP is used to provide service-oriented architecture based on standard HTTP/HTTPS protocol. Due to these improvements and advantages provided by OPC UA, it is widely used in many industrial applications and areas

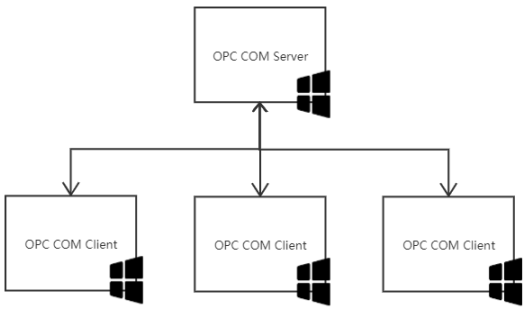

Figure 3: The Architecture of Classic OPC.

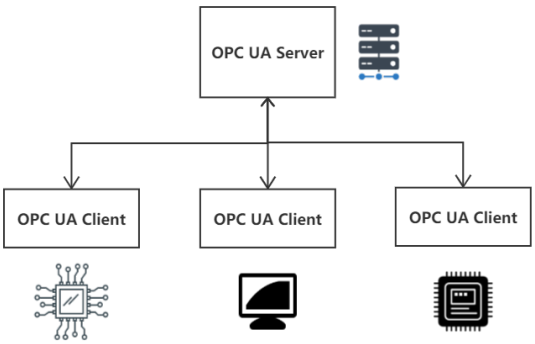

Figure 4: The Architecture of OPC UA.

where the standard OPC specifications were used before. These advantages of OPC UA are shown as follows:

- **Cross-Platform Capability:** Traditional Classic OPC specifications have to run on Windows OS host because the COM/DCOM server functionalities only support Windows OS instead of Linux OS. However, by using SOA and web services, OPC UA is a platform-independent specification and does not require the host's OS. Utilizing the SOAP and XML protocols, OPC UA can work on many devices regardless of host OS. This feature makes OPC UA very popular in industrial areas where many devices with different operating systems are required. Figure 3 and figure 4 present the difference of cross-platform capabilities between Classic OPC and OPC UA.

- **Expanded Security:** The security in OPC UA is enhanced due to eliminating the COM/DCOM used in OPC UA because the COM/DCOM protocols are sometimes problematic. DCOM is used in Classic OPC to provide inter-process security, which is often overlooked by vendors during testing stages. This overlook of inter-process security provided by DCOM often introduces problems to users. The security configuration sometime will be disabled totally in the user-side because of overlook or misconfiguring. Classic OPC developers can only configure the security setting via Access Control lists, while OPC UA

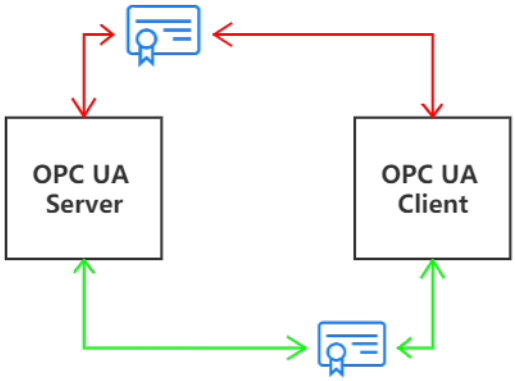

Figure 5: The security mechanism of OPC UA.

developers can use standard web technologies to provide security settings such as authentication and encryption capabilities. Therefore, the COM/DCOM elimination helps OPC UA become widely used and replace the classic OPC specifications based on COM/DCOM. Figure 5 illustrates the scenarios where the OPC UA server and client communicate via unique certificates. OPC UA can provide the X.509 private keys and certificate files by using PKCS12 Public-Key Cryptography standards. The server and client can select both the public keys and the private keys. Three messaging models can be selected by users to communicate between the server and the client. These three modes are: (1) None, (2) Sign, (3) Sign and Encrypt. Moreover, two security policies, i.e., Basic256 and Basic128Rsa15, can be selected by users to sign or encrypt the data between the OPC UA client and server.

- **IT Integration:** OPC UA provides another advantage for developers to easily adapt to existing IT networks because of the standardized security model. OPC UA can communicate through any standard HTTP and UA TCP port, which makes it easy to be used in existing projects using other similar traditional communication protocols. Furthermore, because of this standardized security model, seamless, remote client-server connectivity is allowed for OPC UA through a VPN or firewall. Another feature that makes OPC UA easy-adapted is the standard network protocols, including authentication with certification and data encryption.

- **Compatibility:** OPC UA has changed a lot of its data communication technologies by comparing it with Classic OPC data access models, which means the OPC UA is naturally compatible with Classic OPC DA. For example, if the OPC DA wants to access OPC UA client applications, the UA Wrapper is required for Classic DA. Similarly, OPC DA requires a UA proxy to access the UA server. Figure 6 presents the use case of UA proxy and UA wrapper. NI OPC Servers 2012 and later is one method to bridge DA and UA. The OPC

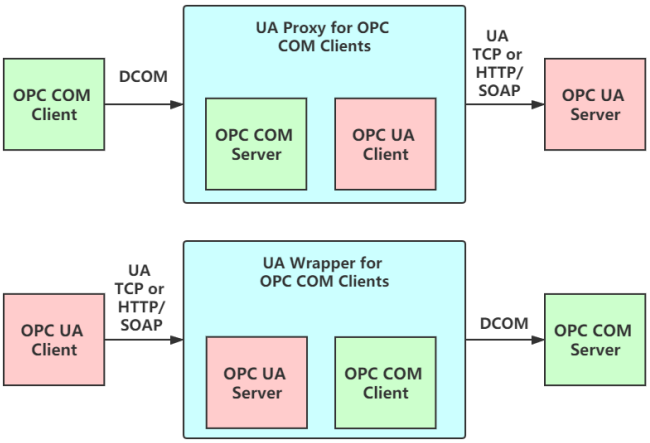

Figure 6: The UA Proxy and UA Wrapper.

UA client driver is provided by the NI OPC servers so that UA client can be republished through an OPC DA server to connect to an NI OPC DA.

2.4 Docker

Docker has been implemented based on containerization technology [22]. Containerization technology becomes more and more popular because of several features it provides:

- **Flexibility:** Most of the applications can be containerized.
- **Lightweight:** The resources, e.g., memory and networking on the host, can be easily shared with multiple containers to make them more lightweight than virtual machines.
- **Portability:** All the images made by developers according to different applications can be shipped to the Docker hub or stored in a local or cloud-based customized Docker registry.
- **Loosely-coupled:** Any container can be updated, deleted, or modified without disrupting other running containers due to the feature of namespaces.
- **Scalability:** Any containerized applications can be scaled by easily indicating the number of replicas via command or the YAML file.
- **Security:** Resources used in different containers are isolated according to the demand for containers. This feature secures both the containers and host.

Docker is open-source software that can be used to build, ship, and deploy containers. The primary technology used in Docker is resource isolation. The Docker resources are isolated from underlying host resources. Cgroups and namespaces are the two main technologies used in the resource isolation feature [23]. Docker has become

famous in container technology, and container technology has been improved a lot due to the development of Docker. Docker becomes popular because of the features provided. The first feature is that containerized applications can be deployed faster than standalone application processes. This feature is also the main goal of this thesis to prove. The second feature is flexibility. Docker allows users to configure many infrastructure components, including CPU, memory and networking via Dockerfile or commands. These two features help companies reduce the time of developing and deploying applications. Multiple containers can run simultaneously on the top of the host. Docker can containerize almost all components of applications. For example, micro-services are the main scenarios using Docker to containerize multiple components and organize them together to perform as a whole application. These containerized components can also be organized and distributed in Docker swarm or Kubernetes orchestration [24].

Images and Containers:

All the files of a container required by Docker to work are bundled in Docker image. The Docker image consists of different layers of necessary files, e.g., runtime, binaries, and environment, of an application. Isolation is one of the essential features that Docker provides to applications so that different containers can run based on their isolated resources and filesystems. The filesystem that each container uses comes from the Docker image of a corresponded container, and this filesystem is different from a container to a container according to specific files, e.g., dependencies and runtime, of their applications.

Containers and Virtual Machines:

With the development of cloud technologies, more and more applications are deployed to the cloud platform, such as Azure, GCP, and AWS. One significant advantage provided by these cloud platforms is the virtual machine. Companies or developers can ask for virtual machines in the cloud as many as they want according to development requirements. Therefore, virtual machines are widely used in many developing and business scenarios. Mainly, virtual machines are extensively used in cloud services like Infrastructure as a Service (IaaS). Other scenarios of widely using virtual machines are that many Platforms as a Service (PaaS) and Software as a Service (SaaS) providers are deploying their applications or services based on a significant number of virtual machines [25]. If all workloads of applications and services are shipping to virtual machines on the cloud, the performance of these services will be limited to virtual machines' performance. Compared with virtual machines, container technology has some unique features, such as scalability, lightweight, and flexibility. Therefore, the container-based virtualization technology becomes an alternative to traditional virtual machines if some features, e.g., scalability and lightweight, are required by services and applications in the cloud [26].

One scenario in which the container technology like Docker can be the perfect alternative is when companies require fast deployment or updates. Because the containers are more lightweight and flexible than virtual machines, the containers can be deployed, scaled, and updated faster than using virtual machines. Containers running on the host can share the host kernel and be isolated because of the advantages of cgroups and namespaces in container technology. Containers are usually more

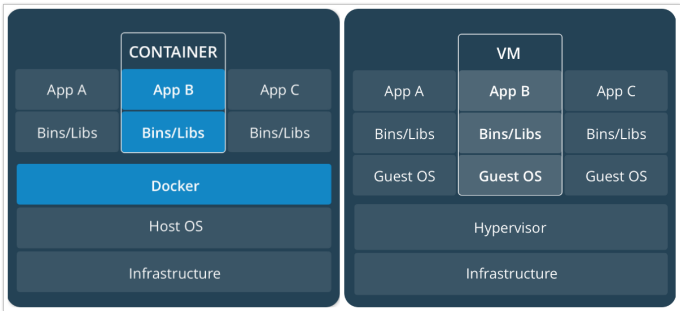

Figure 7: Docker vs. virtual machine [28].

lightweight than virtual machines since containers do not request more resources than needed, but virtual machines request more resources to start and run applications, making virtual machines heavier to start.
Virtual machines use the hypervisor placed on top of the host to coordinate the guest operating systems. Each guest machine has a guest operating system running on top of the hypervisor, making virtual machines more challenging to scale and update. Docker successfully eliminates these difficulties of fast deploying, scaling, and updating applications by utilizing container virtualization technology. This is the reason why Docker becomes more accessible than virtual machines. Figure 7 shows the architectural difference between Docker and Virtual machine [27].
Docker Architecture:
Docker is a container virtualization technology using a client-server architecture where the containers work as clients, and Docker daemon works as a server in the host machine. Containers should communicate with the Docker daemon to run correctly. Specifically, the Docker daemon should receive commands from end-users to request necessary resources from host kernel, pull or build the image, and distribute containers. The Docker daemon and Docker containers can work on the same host or different nodes, depending on the Docker cluster. If Docker has a cluster containing several nodes, the Docker daemon and Docker containers can run separately and communicate via a virtual NIC named Docker0 on the host. Figure 9 shows the Docker architecture. There are several essential components in the Docker orchestration, such as Docker images, containers, and registries. These components will be introduced in detail as follows:

- **Docker Images:** Docker images contain source code necessary to create and run containers. This source code is a read-only template for different images. Each image should contain all the files, e.g., dependencies, binaries, and runtime, required to run the containerized applications. Most popular software images, e.g., Ubuntu, Nginx, and Redis, can be found on the Docker Hub, the public registry for storing Docker images. Developers can either pull the images from the Docker Hub or create their images using the customized Dockerfile. Since there is no image created for the simulator software, Apros used in the experiment of this thesis, there is a need to implement the Dockerfile for that simulator software and build the Apros image. To create a customized image,

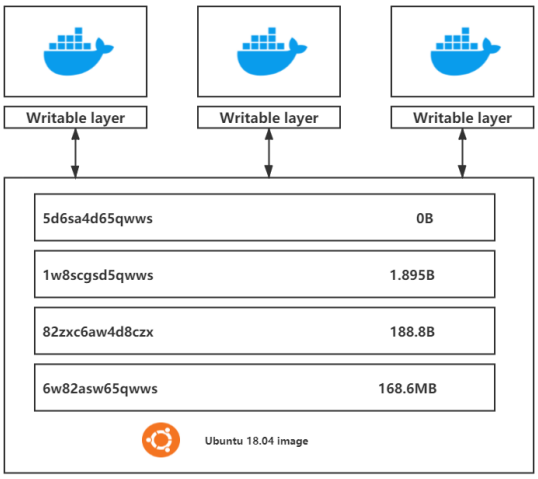

Figure 8: The architecture of Docker images.

developers should carefully write the instructions in the Dockerfile, and the necessary instructions will be introduced in a later section. Each instruction in the Dockerfile will create one layer in the image, and all necessary layers in the image are working together to support the container created from the image. The principle of filesystems in the image is that each layer, according to the instruction in the Dockerfile, will be added on top of existing layers. The layered architecture of Docker images is shown in figure 8. This mechanism of adding layers on top of the existing layers introduces a critical feature in Docker, which is the cache mechanism. The cache mechanism in the Docker means that created layers will be reused in other image-building processes if the same layers are found in the cache. If the developers want to modify the existing code in the image, Docker will copy all the layers and add the changes as a new layer on top of the existing layers. This feature makes the rebuilding image processes more flexible and reusable [23].

- **Docker Containers:** The relationship between the Docker images and the Docker containers is that the Docker containers are the running instances of the corresponding images. The essential difference is that there will be one writable layer created for the running containers on the top of the corresponding images [29]. Other layers are the same among Docker containers built from the same image except for the writable layer on the top. Based on this writable layer, containers can be created from the same image as many as possible. All changes in the writable layer will be eliminated if there is no need to keep these changes as a new image. However, if all the writable layer changes need to be saved as a new image, all layers under the writable layer should be copied first. And then save the writable layer as a new layer on the top of existing layers. This principle is named as "Copy-on-Write" (CoW). Figure 8 presents the writable layers created for containers.
  Each container is running with sufficient isolation, enabling each container to

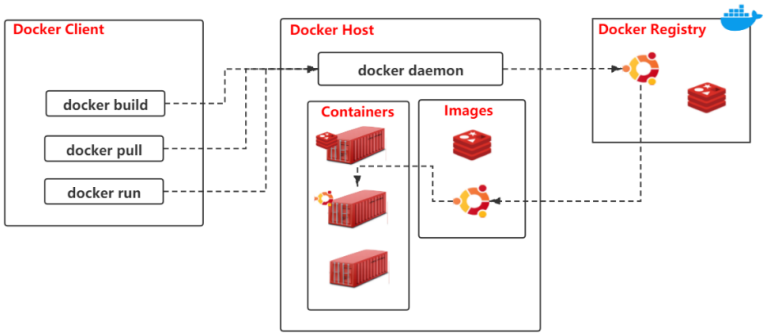

Figure 9: The example of pulling an image from registry.

share the host kernel without interrupting other containers. The customized storages and networks can be created via provided storage drivers and network drivers.

- **Docker Registries:** Docker images can be stored in Docker registries, e.g., Docker Hub and private registries. Docker images developed by developers can be stored in registries so that the developers can easily pull these images. Users with permissions can easily download the images to their laptop to develop. This feature makes the developing process more convenient and faster than the traditional way where the developing environment, e.g., runtime and dependencies, is challenging to set up on another computer quickly. Docker Hub and private registries can work in a way similar to source code repositories [30]. Docker Hub is an official registry provided by Docker, and it is one of the most popular and most significant public registries for storing Docker images. By comparing with public registries, most of the time, developers need to control their access rights of images. This is the scenario of using private registries where the access rights can be configured and controlled by developers according to different purposes.

- **Docker Client:** Users can interact with Docker daemon using the Docker client. Docker platform is working on a client-server architecture. The Docker client is a command-line interface (CLI) where the end-users can send Docker command via the CLI, such as Docker run, Docker build, and Docker pull. These commands will be sent to Docker daemon so that the Docker daemon can explain and execute these commands sent from users. Additionally, users can send commands to Docker daemon via the REST API of Docker daemon, which makes communication between Docker clients and Docker daemon easy and efficient.

- **Docker Engine:** The relationship among Docker daemon, REST API, and CLI is shown in figure 10.

  – The CLI is used by end-users to send Docker commands.

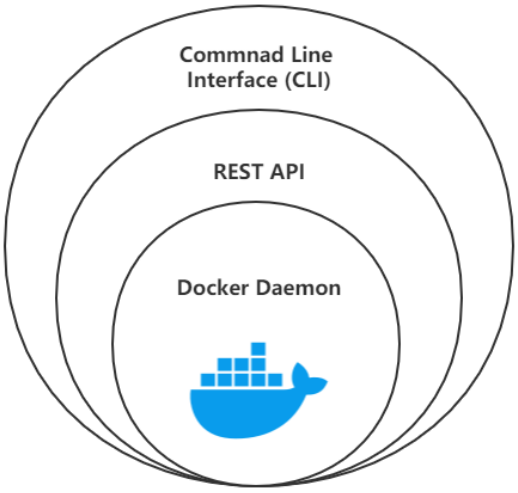

Figure 10: The architecture of Docker engine.

  - The REST API specifies interfaces that can be used to interact with Docker daemon.
  - Docker daemon manages Docker containers on the host system. The Docker commands will be sent from Docker REST API to Docker daemon so that the daemon can manage Docker objects such as images, containers, and networks.

- **Dockerfile:** The Dockerfile is a text file that stores all the instructions used to build the Docker images. According to different applications, different instructions can be used to create different Docker images, and users can use the command, Docker build, to build the customized Docker images. These instructions in the Dockerfile define the application environments, such as dependencies, binaries, and runtime. After executing the Docker command to build the Docker image, the Docker engine will go through all the instructions in the Dockerfile and execute them in the order specified. The cache mechanism will be applied during the process of building an image from Dockerfile. Specifically, if the new layer, according to an instruction in the Dockerfile exists, the Docker engine will reuse this cache layer instead of creating a new same layer. The order of executing the instructions in the Dockerfile must start from the instruction ('FROM'), which indicates the base image of this new image. Furthermore, the base image can also be an empty filesystem started by Docker if the base image is SCRATCH.

2.5 Container Orchestration

Last section introduces the features of container virtualization and Docker in detail. Docker makes containers easy to be packaged, ported, and distributed. Docker is working on a client-server architecture where the client and server could work on

different nodes. Nodes and containers can be easily scaled up and down due to the features like scalability and flexibility. However, this scalability feature introduces one problem: if the number of nodes and containers is great, these containers are challenging to be managed by Docker. Furthermore, multiple containerized micro-service applications are usually deployed on different nodes, and these containers need to communicate with each other [31]. The communication between multiple containers on different nodes is very difficult to achieve if the Docker is the only tool to manage the containers. Docker swarm and Kubernetes help to achieve this motivation of managing the containers automatically in a distributed way. Container orchestration such as Docker swarm and Kubernetes can be used to set up clusters with nodes, run containers, create networks, and update containers automatically. This section introduces the container orchestration of the Docker swarm and Kubernetes in detail.

2.5.1 Docker Swarm

In the Docker swarm, there will be several nodes working as the managers or the workers. According to different situations, these manager nodes and worker nodes can run on different operating systems such as Linux and Windows [32]. Moreover, after creating the Docker swarm with multiple nodes, nodes in the cluster can leave at any time, and other nodes can join to this Docker swarm cluster by fetching the token from manager nodes easily. This flexibility makes the Docker swarm cluster easy to distribute containers easily among all the nodes in the cluster. After establishing the Docker swarm cluster with nodes meeting the requirements, Docker services can be created according to different settings, including the number of replicas, memory, or CPU requirements and networks.
Moreover, if there is a need to change the Docker service settings, these changes will be effective immediately without restarting any containers or services. The Docker host will automatically create/delete related containers to automatically meet the new Docker service settings' configuration. Therefore, when the container environment is a compound, or many containers need to be organized, the Docker swarm can be a useful option. Figure 11 presents the architecture of Docker Swarm nodes. Docker swarm can orchestrate containers in a distributed way, and the Docker swarm mode can be used directly in Docker-engine v1.12 and later. Because of the Docker engine and Docker swarm integration, the Docker CLI can also be used by Docker swarm commands and the Docker REST API. Therefore, the Docker swarm cluster can be set up quickly with several commands from Docker CLI, making the Docker swarm very competent in the container orchestration areas [24]. Some critical components of Docker swarm will be introduced as follows:

- **Nodes:** One or multiple nodes running on Docker swarm mode constitute to the Docker swarm cluster. The nodes in the Docker swarm cluster can have different roles, for example, workers and managers. Different roles have different capabilities. For example, if one node wants to join the existing cluster, the token used to join can only be achieved from manager nodes with proper commands. Manager nodes in the cluster should be mainly responsible for managing the cluster and scheduling the services with proper configurations.

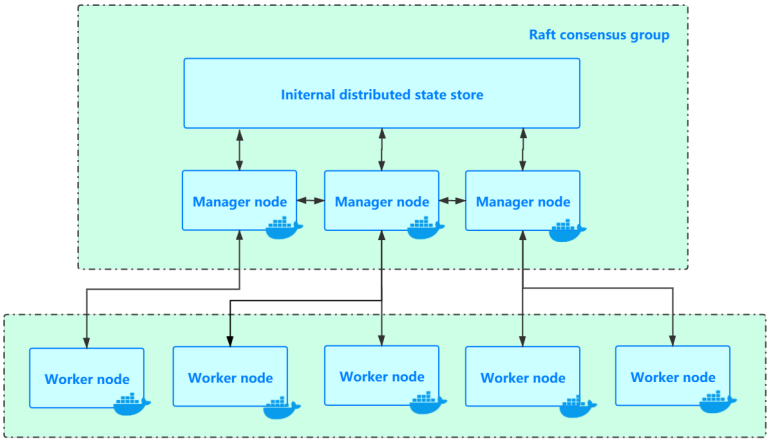

Figure 11: Architecture of Docker Swarm nodes.

The Docker service configurations will be sent to one of the manager nodes so that the manager nodes received the service request will prepare the service environment according to the configuration, such as network, memory, and CPU. After preparing the customized environment for the service, the manager nodes will schedule related containers in worker nodes that receive the requests from manager nodes and provide sufficient resources to fulfill the service requirements. In the Docker swarm cluster, fault tolerance can be achieved by having multiple manager nodes, because manager nodes are responsible for maintaining the Docker swarm cluster. A single leader among manager nodes should be selected if there is more than one manager in the Docker swarm cluster. Other manager nodes that are not the leader are always ready to be selected as the new leader if the previous leader is lost. In the Docker swarm cluster, the manager nodes can maintain the cluster while the worker nodes are responsible for executing the workloads received from manager nodes. However, the workloads from one manager node can also be re-directed to other manager nodes to be executed. The mechanism of scheduling workloads is defined by the scheduler working in the leader node. Whether the node can receive the workloads or not depends on three statuses of the node, which are Active, Pause, and Drain.
The node in the Active status is always ready to receive workloads from the leader node. The node in the Pause status will stop receiving new workloads from the leader node, but the existing workloads will continue to be executed in the paused mode. The node in the Drain status will not receive the new workloads from the leader node while existing workloads will be re-scheduled to other active nodes. Besides using the nodes' status to decide which nodes should receive the workloads, the nodes can also be selected via the node labels so that the node with the matched label will receive the new workloads from manager nodes [33].

- **Service:** Docker swarm services indicate the specific number of containers with settings in the service, and these containers will be created and scheduled

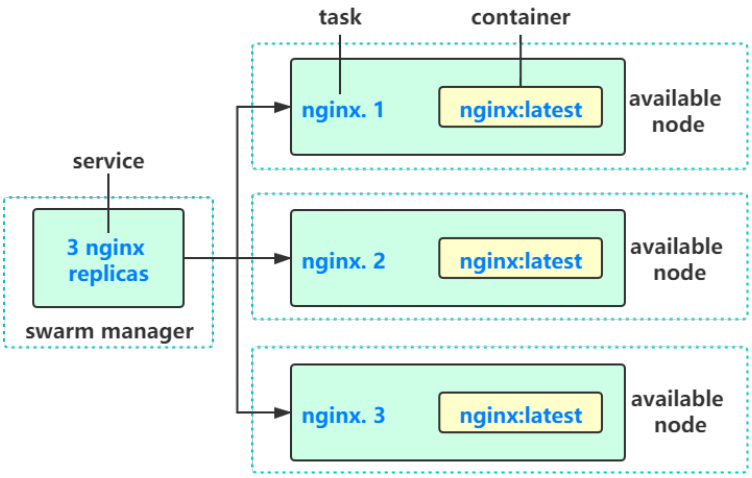

Figure 12: An example of swarm service for creating three Nginx containers.

to worker nodes. Additionally, users can create multiple containers in worker nodes through swarm services with customized settings, including the number of replicas, network settings, images, and the supported operating systems. All the settings with Docker swarm service commands will be sent to the manager nodes where the service configurations will be parsed and handled separately. For example, if swarm service defines the replicas of the container, then the manager nodes will try to ensure the status of the containers meeting the requirement. The figure 12 presents the example.
Based on this mechanism, if there is no container lost, the manager nodes will re-schedule another new container to replace the lost one. Two types of container-scaling modes can be defined in the Docker swarm, which are global and replicated. The global mode means there will be one container in each node in the cluster, while the replicated mode means that the number of replicas can be defined in the swarm service. Multiple containers with the same image may run on the same node [34].

- **Task:** Docker swarm tasks cooperate with swarm services to meet customized applications' needs. Specifically, swarm services configure the swarm applications while the tasks are used to do the work based on the configurations or settings defined in swarm services. In other words, tasks are execution units running until completion. If a task stops, the task will not be executed again, but a new task will be created, replacing the stopped task. Every task will go through several states forward but not backward until they complete or fail. These states that the tasks will go through are presented as follows:

  - NEW: Initialized task.
  - PENDING: Resources were allocated for the task.
  - ASSIGNED: The task was assigned to the node by Docker.

- ACCEPTED: The worker node accepts the task.
- PREPARING: Docker is preparing the task.
- STARTING: Docker is starting the task.
- RUNNING: The task in under execution.
- COMPLETE: The task completed successfully.
- FAILED: The task failed.
- SHUTDOWN: The task will be shut down.
- REJECTED: The work node rejected the task.
- ORPHANED: The node was lost for too long.
- REMOVE: The task was removed due to removed associate service.

2.5.2 Kubernetes

More and more applications and components in Google are containerized with container virtualization technology, and these Google application containers are running in Google's data centers [35]. Moreover, the developing trend in Google is that more applications will be shifted from the traditional developing style to a containerized style. Google manages its containerized applications via an internal container cluster management system named Borg. Since the container virtualization technology becomes more and more popular, Google introduced its container orchestration tool called Kubernetes based on the Borg container management system [35].
Kubernetes has been developed as an open-source product for orchestrating, managing, and scaling containers [36]. Several critical components of Kubernetes are presented below.
Master and Worker Nodes: Kubernetes components consist of master components and node components. The nodes in the Kubernetes cluster are worker machines that execute the workloads and run pods from master nodes. Usually, all the master node components are running together on the same node instead of being deployed distributively. Nodes and masters constitute the Kubernetes clusters. Masters provide the control plane of the Kubernetes cluster. Different master components are shown in figure 13, and these components will be introduced in detail in the later section.

- The Kubernetes API is exposed by an API server that is a critical component of the Kubernetes control plane. The API server is the front end of the Kubernetes control plane. The kube-apiserver is the primary implementation of the Kubernetes API server. The design principle of the kube-apiserver is that the kube-apiserver should be able to scale horizontally, which means the scalability of the kube-apiserver can be achieved by increasing the number of instances. Several kube-apiservers can work simultaneously, but the workloads traffic should be balanced among these kube-apiservers. Due to the Kubernetes API server, developers can interact with the Kubernetes cluster through either the REST API calls or the CLI tool, e.g., kubectl. Furthermore, the Kubernetes

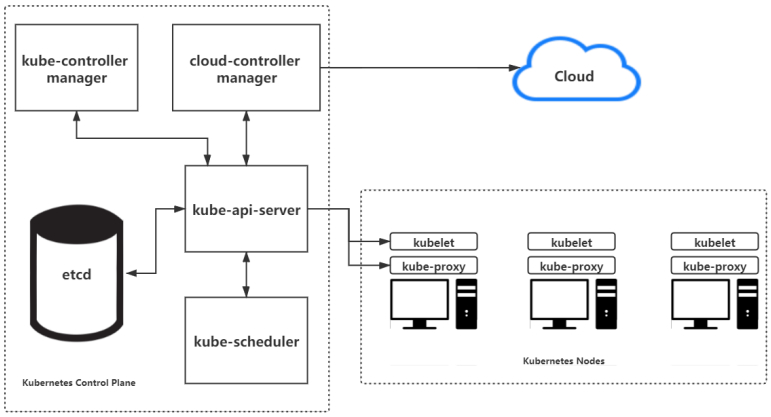

Figure 13: The architecture of Kubernetes.

API server will also handle all the communications between masters and workers in the cluster.

- The controller managers will manage controller processes in the Kubernetes control plane. Controller processes are control loops on the master. They check the state of the cluster via the Kubernetes API server and make the Kubernetes objects move to the desired state. Many controller processes are running on the master nodes according to different functionalities. For example, the replication controller process checks the number of the replicas of pods, and if the number is not the number desired, the controller process will try to make the number of pods reach the desired number. The controller should be working as multiple processes on the master. However, in order to reduce the complexity, all controller processes are compiled into a single binary and run in a single process on the master. All the controllers are:
  - **Node Controller:** Handling issues when nodes go down.
  - **Replication Controller:** Maintaining the desired number of pods for every replication controller object.
  - **Endpoint Controller:** Populates the Endpoints object.
  - **Service Account & Token Controller:** The default accounts and API access tokens will be created for new namespaces.
- The kube-scheduler is an essential Kubernetes control plane component that can monitor the newly created pods. If the newly created pods are not scheduled and working successfully, the kube-scheduler will schedule the pods to proper nodes. Some facts need to be considered when scheduling pods to nodes, including hardware or software constraints, individual and collective resource requirements, data locality, affinity and anti-affinity specifications, and deadlines. These facts will be considered according to the desired configurations of pods.

- The etcd is consistent and highly-available, which stores key-value as Kubernetes' backup store for all cluster data. A backup plan should be designed and applied if the etcd is applied in the Kubernetes cluster as a backing store.

- The cloud-specific control logic is embedded in the cloud-controller-manager. The cloud provider's API can be linked to the Kubernetes cluster through the cloud controller manager. The cloud-controller-manager will only manage the controllers related to your cloud providers, e.g., GCP, AWS, and Azure. If the Kubernetes cluster is running on-premises, there is no need to configure the cluster's cloud-controller-manager. The cloud-controller-managers have many independent control loops logically similar to the kube-controller-managers, and these control loops will be combined into one single binary working as a single control process for reducing the complexity.
  The controllers with the cloud provider dependencies are shown as follows:

  - **Node Controller:** The node controller is used to check if the non-responding nodes are deleted in the cloud or not.
  - **Route Controller:** The route controller sets up the route policy in the cloud.
  - **Service Controller:** The service controller is used to create, update and delete cloud provider load balancers.

- Cluster features are implemented by Add-ons using Kubernetes resources, e.g., DaemonSet and Deployment. Add-ons extend the functionalities of Kubernetes. There are several add-ons are introduced below:

  - **Web UI (Dashboard):** This tool is for managing the Kubernetes based on a Web UI.
  - **DNS:** All Kubernetes clusters should be configured with the cluster DNS add-on because many Kubernetes components rely on it.
  - **Container Resource Monitoring:** Container Resource Monitoring records generic time-series metrics about containers in a central database with a UI used to browse the records.
  - **Cluster-level Logging:** Cluster-level Logging is a mechanism used to save container logs to a central log store provided with a UI to browse.

- Kubelete is an agent running on each node in the Kubernetes cluster. Kubelet ensures that all containers can work in a Pod. Pods and containers in the nodes are managed by kubelet so that these pods and containers can work correctly and healthy. Containers or pods running on the nodes can be created, monitored, and deleted according to the instructions that the kubelet receives. Correctly, the kubelet works according to the configuration in the PodSpec, which is a document in YAML or JSON format. The PodSpec document defines the desired states of the pods running in the nodes.

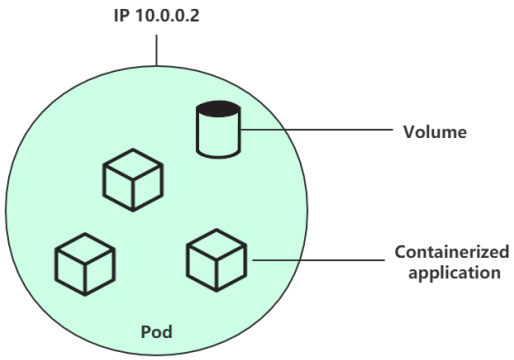

Figure 14: One simple example of a pod.

- Kube-proxy provides a network proxy mapping the individual containers to services, and also it can provide load balancing features on each node in the cluster. Kube-proxy maintains network rules enabling network communication from inside or outside network sessions to the pods in the cluster.

Pod: In a pod, there could be one or more containers. The pods are the basic building blocks in Kubernetes architecture. These containers in the same pod can share storage/network, IP address, port space, and the specification of desired states of all containers in the pod. Each pod in the cluster will receive one unique IP address, and this IP address is shared among all containers in the same pod. For the containers in the same pod, this pod is similar to the host, while containers are similar to the applications or processes running in the host. Therefore, all containers running in the same pod can communicate with each other as applications and processes communicate in the localhost. Compared to the containers in the same pod, containers in different pods can also communicate with each other via the pod IP address. Pods are considered to be relatively ephemeral entities which are similar to the individual containers. The lifecycle of pods is that pods are created, assigned a unique ID (UID), and scheduled to selected nodes, and the pods remain until the nodes go down or these pods are deleted. Based on the feature of pods lifecycle, pods are like ephemeral entities rather than durable entities. One simple example of a pod is shown in figure 14.
Replication Controller and Replica Set: Usually, multiple duplicated pods will be created and run simultaneously to handle the vast workloads and fault tolerance. The number of replicas indicates how many duplicated pods will be created and run at the same time. The replication controller is used to control the number of replicas of pods based on a set of rules defined in a pod template. If the number of pods is lower or more than the desired number in the template, the replication controller will start or terminate pods to meet the desired number. Pods will not be deleted if the related replication controller is deleted on purpose. The replication controller is very competent and helpful in maintaining the pods, and for this reason, it is suggested to use the replication controller even only one pod is required.
Service: The Kubernetes service works with the replication controller to ensure all

replicated pods or single pods are accessible from outside networks. The service in the Kubernetes is an abstraction that is different from the service in the Docker swarm. The Kubernetes service defines a set of pods and a policy for these pods to be accessed externally. Although each pod has a unique IP address, the containers in the pod can not be reached without utilizing the service for the pods. Kubernetes services enable applications in the pods to receive traffic from the outside world. By specifying the type in the ServiceSpec, different services can be exposed and shown as follows:

- **ClusterIP (default):** Service will be exposed to an internal IP in the cluster. By using this type of service, the service can only be reached within the same cluster.
- **NodePort:** Service will be exposed to the same port selected on every node in the cluster using NAT. The parameter <NodeIP>:<NodePort> makes the service accessible from outside the cluster.
- **LoadBalancer:** The external load balancer will be created in the current cloud platform with an external IP to the service.
- **ExternalName:** An ExternalName is to expose the service by returning a CNAME record with the name.

3 Research Approach

In order to introduce the experiment of this thesis, three main parts need to be discussed. The first part is about the general test design and implementation methodology in section 3.1. The second part in section 3.2 is about the general description of the experiment. The last part in section 3.3 introduces all the necessary testing tools and hardware configurations of the experiment.

3.1 The Experiment Design and Implementation

- **Testing the Dockerfile for the Apros**
  Because the official version of Apros image or Dockerfile is not available in the Docker Hub, Apros Dockerfile and Apros image should be created manually. Therefore, the first step of this experiment is to create the Apros Dockerfile. And then test the Apros container in Docker swarm mode according to different aspects, for example, the scalability and the stability. This testing for the Apros container is critical because Apros running inside the container is the version without UI so that SCL scripts have to be used to control the Apros. Therefore, before moving to test Apros container in Docker swarm or Kubernetes, the Dockerfile for building the Apros image and the Apros container built from the image should be tested first.

- **Testing the Communication between Two Apros Instances**
  For this experiment of communication of two Apros instances, 500 testing cases will be used both in Apros1 and Apros2. The Apros1 and Apros2 are the two Apros instances running with the process mode and control mode, respectively. These 500 testing cases are split into two parts (signals and setpoints), and each of the parts is sent to one Apros instance. Therefore, the general design of this experiment is that there are 500 duplicated testing units, and they are used to test the communication stability between two Apros instances via OPC UA interfaces. The structure and configuration of OPC UA configuration files, e.g., ServerConfig.xml and ConnectionConfig.xml, will be tested in this step. If the data in two Apros instances can be exchanged correctly, these OPC UA configuration files are proved to be configured correctly. Importantly, these two Apros instances are two Apros processes running on-premises rather than two Apros containers. The OPC UA configuration files are also tested for these two local Apros processes. However, the OPC UA configuration files will be almost the same for Apros processes on-premises and Apros containers in the cloud. Therefore, once the OPC UA configuration files work as expected, these configurations files can be used in the Apros containers directly only with some minor changes, e.g., IP addresses and names.

- **Creating the Docker Swarm and Kubernetes Cluster on GCP**
  The Docker swarm cluster and the Kubernetes cluster should be created on GCP so that the Docker engine can work on multiple nodes in the cluster. Because the Docker swarm and the Kubernetes will be used to orchestrate

simulation containers distributed to the nodes in the clusters, these two clusters should be created according to different settings. For example, the OS of the master nodes in the Docker swarm cluster could be either Linux or Windows, but the master nodes in the Kubernetes cluster only support Linux rather than Windows. Therefore, A Linux-based VM should be created to behave as the master node in the Kubernetes cluster, while nodes in the Docker swarm cluster can only run with Windows OS. After creating these two clusters on GCP, a simple image will be used to test if all the required runtime and environment is configured correctly so that containers can run in these two clusters successfully.

- **Implementing the Kubernetes Java Client**
  The Kubernetes Java client will be implemented by utilizing the Kubernetes Java API libraries so that the Kubernetes API server can receive commands from the Java application. In this Java application, several methods are implemented to operate and interact with the Kubernetes cluster. For example, one method is named createPod() used to create one or more pods with simulation images. Moreover, methods, including deletePodByID(), getPodStatus() and getPods(), are also implemented for different purposes. Kubernetes API receives the commands from the Java method with a REST API call format. The testing case for creating the simulation pod is a deployment.yaml file. This testing deployment.yaml file will be created, and specifications in the YAML file will be used to create a testing container in the Kubernetes cluster. If such a pod can be viewed in one of the nodes in the cluster, this Java application is proved to be implemented correctly to interact with the Kubernetes cluster.

- **Testing the Containerized Apros in the Docker Swarm Mode**
  The Docker swarm cluster will be tested first by running simulation containers in the cluster. Because the Docker swarm cluster orchestration is not complicated, the experiment for testing the Docker swarm cluster creation is simple. The container with windows server image will be asked to run on a Windows node in the cluster. If this Windows container can be created and work in the Docker swarm cluster, this cluster is proved to be created successfully, and it is ready to support the simulation containers with Windows base images.

- **Testing the Containerized Apros in the Kubernetes Mode**
  This step is to test the Apros container in Kubernetes mode. By comparing with running the Apros container in Docker swarm mode, running Apros container in Kubernetes is very different even though the same Apros image should be used in this experiment. In the Kubernetes mode, different yaml files need to be configured before running Apros containers. For example, one of the critical YAML files is the deployment.yml, which configures the most features of running pods. After successfully testing the Apros container in Kubernetes mode, several aspects should be collected and analyzed to fulfill the design of this experiment and achieve the main object of this thesis. These collected results should be analyzed mathematically to prove some features, e.g., scalability and stability.

- **Evaluating the Performance Between Containerized Simulators and Non-containerized Simulators**
  After testing the Apros containers both in Docker swarm and Kubernetes mode, it is time to finally test the Apros containers' communication via OPC UA interfaces. This communication testing is the main object as we designed for this thesis. This experiment compares different aspects like the scalability and stability between standalone Apros instance and Apros containers.
  One of the most important features provided by Docker is the scalability. The scalability is also the reason why the Docker container technology is used for Apros or other simulators. Therefore, the Apros containers' scalability will be tested, and the expected result is that the Apros container performs better than standalone Apros instance when concerning the scalability. The experiment method is that an Apros container and standalone Apros instance will be assigned the same workloads, and then the Apros container and standalone Apros instance will be scaled to two. By measuring the duration time and some other features, e.g., flexibility and convenience, the scalability can be measured.

3.2 Description of the Tests

The design and implementation need to be tested in order to achieve the goals set in the introduction section. These tests are shortly described in this section. The objective of the tests is to ensure that the core functionalities of the implementation operate appropriately. The more detailed descriptions of the tests, including the results and discussion, are presented in chapter 5.
The main motivation of the work is to prove that the co-simulation mechanism could perform better than standalone simulation software instances. Specifically, the objective of the experiment in this thesis is to implement a co-simulation platform integrated with some components, e.g., the Kubernetes Java application and OPC UA-configuration-file generator. After implementing the co-simulation with all the designed components, the multiple cooperative containers running in the co-simulation platform can be compared to the standalone simulation software processes. By collecting and analyzing the experiment results, e.g., the execution time, flexibility, and convenience, the co-simulation mechanism's scalability can be evaluated and proved.

3.3 Testing Tools

This section presents the most important tools used in the experimental part of this thesis. In addition to the software, the hardware settings are presented to help to understand and evaluate the testing results. It should be noticed that different versions of Docker and Kubernetes may result in different results because Docker and Kubernetes update frequently.
Docker version:

- Client: Docker Engine - Community, Version: 19.03.11, API version: 1.40, Go version: go1.13.10, Git commit: 42e35e61f3, Built: Mon Jun 1 09:12:41 2020, OS/Arch: linux/amd64, Experimental: false, Server: Docker Engine - Community
- Engine: Version: 19.03.11, API version: 1.40 (minimum version 1.12), Go version:, go1.13.10, Git commit: 42e35e61f3, Built: Mon Jun 1 09:11:14 2020, OS/Arch: linux/amd64, Experimental: false, Version: 1.2.13, Version: 1.0.0-rc10, Version: 0.18.0, GitCommit: fec3683

Kubernetes version:

- Client Version: version.InfoMajor:"1", Minor:"18", GitVersion:"v1.18.3", GitCommit:"2e7996e3e2712684bc73f0dec0200d64eec7fe40", GitTreeState:"clean", BuildDate:"2020-05-20T12:52:00Z", GoVersion:"go1.13.9", Compiler:"gc", Platform:"linux/amd64"
- Server Version: version.InfoMajor:"1", Minor:"16+", GitVersion:"v1.16.9-gke.2", GitCommit:"4751ff766b3f6dbf6c6a63394a909e8108e89744", GitTreeState:"clean", BuildDate:"2020-05-08T16:44:50Z", GoVersion:"go1.13.9b4", Compiler:"gc", Platform:"linux/amd64"

Docker swarm windows worker node:

- Machine type: Machine type: n1-standard-2
- OS: Windows Server 2019 Datacenter
- CPU: 2 vCPU
- Memory: 7.5 GB memory

Kubernetes Linux manager node:

- Machine type: n1-standard-1
- OS: Ubuntu 18.04 LTS
- CPU: 1 vCPU
- Memory: 3.75 GB memory

Kubernetes windows worker node:

- Machine type: n1-standard-2
- OS: Windows Server 2019 Datacenter
- CPU: 2 vCPU
- Memory: 7.5 GB memory

4 Implementation

Chapter 4 presents the methods and configurations about how to implement the designed experiment in detail. Specifically, section 4.1 introduces the architecture of the Docker swarm and Kubernetes cluster. Section 4.2 gives the Dockerfile used in this thesis to build the Apros image. Section 4.3 introduces the Java application implemented in this experiment to automatically generate OPC UA configuration XML files by receiving specific configurations from end-users with SCL files. Section 4.4 presents the Java client, which is used to deploy Kubernetes pods via Kubernetes Java commands so that the deployment of simulation containers can be deployed automatically by receiving commands from the upper level. Figure 15 presents the orchestration of Docker swarm.

4.1 Architecture of Docker Swarm and Kubernetes Cluster

There are three virtual machines created on Google Cloud Platform with the name manage1, worker1, and worker2. The manager1 runs with Linux OS while the worker1 and worker2 run with Windows OS because the simulation software used in this experiment is only supporting Windows OS.

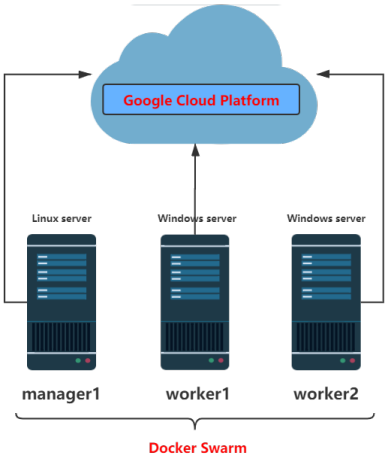

Figure 15: The orchestration of Docker swarm.

Figure 15 shows the basic orchestration of the Docker swarm cluster on GCP. Due to the release of Kubernetes 1.14 on Windows server 1809, several new features were introduced with this release:

- **Overlay Networking:** A virtual overlay network should be configured and implemented using Flannel in vxlan mode.
- **Simplified Network Management:** The route management between nodes is automatic using Flannel in host-gateway mode.

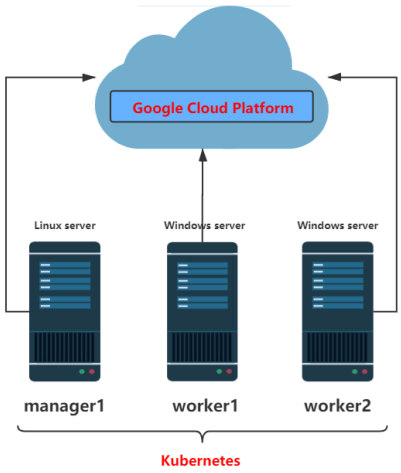

Figure 16: The orchestration of Kubernetes.

- **Scalability Improvements:** Due to the advantage of deviceless vNICs for Windows server containers, the windows containers can be deployed faster and more reliable.
- **Hyper-V Isolation:** Hyper-V isolation provided by the Windows server makes containers running on Windows servers secure.
- **Storage Plugins:** The FlexVolume storage plugin is supported as a storage plugin for windows containers.

The architecture of Kubernetes is a master-worker mode where the Kubernetes control plane runs on the master node. It should be noticed that the master node only supports the Linux OS rather than Windows OS. Currently, it is impossible to have a Kubernetes cluster with only Windows servers. This is the reason why the Windows nodes should be configured as worker nodes in the Kubernetes cluster. The orchestration of Kubernetes is very similar to the orchestration of the Docker swarm, where there are three VMs created for the test on GCP. Moreover, two Windows servers are created for running the simulation software later. The orchestration of the Kubernetes cluster is shown in figure 16.

4.1.1 Steps to Create the Docker Swarm Cluster

By comparing with the Kubernetes, the Docker swarm is much easier to create. The manager node can be running on Windows OS directly.
The parameter of –advertise-addr is used to configure the publish address used by the manager node. This manager publishing address must be accessible by other nodes to be able to join the existing cluster.
After initializing the Docker swarm and creating the manger node successfully, the worker nodes can be added to this existing Docker Swarm now. The first step to

join a node to the existing Docker swarm cluster is to get the join-token. There are two methods to get this token, which are either obtaining the token from the output of creating the cluster or obtaining the token by the command in the master nodes. The final step to joining a node to the created Docker Swarm is to copy the joining command with the token on the node that is waiting to be connected to the cluster. By executing the same command on different nodes, many workers can be added to the existing Docker Swarm cluster. Moreover, the joining node's role can be defined as a worker or manager with different joining tokens. Docker's mechanism makes the Docker Swarm cluster more competent with different operating systems and different roles so that this cluster can be more flexible and scalable.

4.1.2 Steps to Create the Kubernetes Cluster

Because at least one Windows server should be added into the Kubernetes cluster, the mixed-OS Kubernetes cluster configuration is more complicated than the Linux-only Kubernetes cluster. In this section, the critical steps of implementing the mixed-OS Kubernetes cluster on GCP will be illustrated.
Creating a Kubernetes master:

- **Install Docker:** To run containers in the Kubernetes cluster, the container engine, e.g., Docker, needs to be installed first on the server.
- **Install Kubeadm:** The kubeadm is a tool used to create the Kubernetes cluster, and it also should be installed firstly on the Linux server because the manager node in the Kubernetes should be Linux-based node.
- **Initialize Master Node:** After initializing the master node, the cluster subnet and service subnet should be kept for further configuration.
- **Enable Mixed-OS Scheduling:** By default, the source code of Kubernetes makes all Kubernetes resources duplicated on all nodes in the Kubernetes cluster. However, the scenario in this test is that the Kubernetes cluster is a mixed-OS cluster. Therefore, the NodeSelector labels should be applied to configure the specific nodes based on their OS.
- **Collect Cluster Information:** After doing the above steps, critical information should be kept for joining future Windows nodes.

Joining Windows Server Nodes to A Cluster:

- **Install Docker:** It is the same as configuring the Linux master node where the Docker engine should be installed on the Windows node for running containers on it later.
- **Create the Infrastructure Image:** The infrastructure image nanoserver:1809 should be pulled and prepared on the Windows nodes in this step.

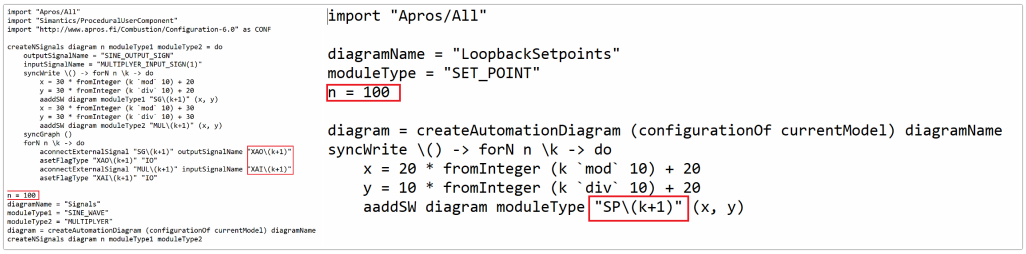

Figure 17: The SCL scripts used in the headless-version Apros.

- **Download Kubernetes Binaries:** Install the Kubernetes on the Windows nodes. To run Kubernetes, several tools also need to be installed with Kubernetes, kubectl, kubelet, and Kube-proxy binaries.
- **Join the Windows Nodes:** Two critical steps need to be done to join the Windows nodes to the Kubernetes cluster. The first step is to join the Windows nodes to a Flannel (vxlan or host-gw) cluster. And, the second step is to join the Windows nodes to a cluster with a ToR switch.

4.2 Build Apros Image via Dockerfile

Because there is no official version of Apros image that can be found in the public registries, the Apros image should be created via a customized Dockerfile. The Dockerfile in Appendix A shows the steps to build the Apros image, including the working directory and default launching command. All the runtime, binaries, and environment configurations are indicated in this Dockerfile to ensure that all of them will be added to the Apros image. After configuring the Dockerfile, the Apros image can be built from it. It should be noticed that the Apros is not available for free publicly. Thus these instructions in the Dockerfile indicate that all necessary files of the installed Apros on the local machine with permission and license should be copied to the Docker image.

4.3 Generate OPC UA Configuration Files

Because the scenario used in this experiment is that multiple simulators are used to provide the better performance from cooperative simulators, the configuration of OPC UA interfaces should be dynamic according to different specifications among all simulators in real-time. This scenario requires the OPC UA configuration XML files to be configured automatically in real-time. Therefore, one Java application was implemented in this thesis to automatically create the required OPC UA configuration files by receiving the required data, e.g., the number of OPC UA connections from SCL scripts. Figure 17 presents the testing case that the specifications, e.g., number of connections (n = 100) and data type (SP), are processed by the Java application and the results of generated OPC UA configuration file according to these two SCL scripts are presented in figure 18. Each DxConnection indicates one connection from the source item to the destination item. For example, the source item in the

```
<DxConnection
    DxItemId="DX.DXConnectionsRoot.Signals Server.toLoopbackSetpoints"
    DxItemName="from Signals.XAO1" Scale="1.0"
    SourceItemId="TYPES!A!ANALOG_SIGNAL!XAO1.ANALOG_VALUE"
    SourceItemName="" SourceItemPath=""
    SourceNamespaceIndex="2"
    TargetItemDataType="Double"
    TargetItemId="TYPES!S!SET_POINT!SP1.SP_VALUE"
    TargetItemName="" TargetItemPath="" TargetNamespaceIndex="2">
    <BrowsePath>Signals Server</BrowsePath>
    <Description/>
    <DefaultOverrideValue/>
    <SubstituteValue/>
    <VendorData/>
</DxConnection>
<DxConnection
    DxItemId="DX.DXConnectionsRoot.Signals Server.toLoopbackSetpoints"
    DxItemName="from Signals.XAO2" Scale="1.0"
    SourceItemId="TYPES!A!ANALOG_SIGNAL!XAO2.ANALOG_VALUE"
    SourceItemName="" SourceItemPath=""
    SourceNamespaceIndex="2"
    TargetItemDataType="Double"
    TargetItemId="TYPES!S!SET_POINT!SP2.SP_VALUE"
    TargetItemName="" TargetItemPath="" TargetNamespaceIndex="2">
    <BrowsePath>Signals Server</BrowsePath>
    <Description/>
    <DefaultOverrideValue/>
    <SubstituteValue/>
    <VendorData/>
</DxConnection>
```

Figure 18: The example of OPC UA ConnectionConfig.xml file.

first DxConnection is the XA01.ANALOG_VALUE and the destination item is the SP1.SP_VALUE. This DxConnection means the data of XA01.ANALOG_VALUE will be transmitted to the SP1.SP_VALUE so that these two items in two different Apros instances should be the same. The experiment results are shown in section 5. There are two primary considerations for implementing this Java application to generate the OPC UA configuration XML files automatically. The first reason is that this Java application can generate a significant number of configuration files for multiple simulators in rea-time. It is even impossible for developers to manually prepare for these files for more than 100 connections in a short time. By implementing this Java application, all configuration XML files required by OPC UA can be provided almost immediately for all Apros instances. The second consideration is the number of Apros instances. If the number of Apros instances is 10 or 20 with different OPC UA server addresses and requirements, it is also impossible to manually provide the required configuration files quickly and correctly. These two reasons are the main motivations of implementing the Java application that can automatically abstract the useful data and generate the required configurations files for all Apros instances.

4.4 Deploy Apros Pod in Kubernetes by Java Client

Kubernetes provides the REST API for users to interact with Kubernetes core components. All user commands, e.g., kubectl get nodes, and operations and communications between Kubernetes components are handled in a format of REST API by Kubernetes API server. Therefore, all components in the Kubernetes cluster can be interacted in the REST calls format by external users. There are two main methods for users to operate the Kubernetes cluster. The first method is to use the Kubernetes tools, such as the kubectl and kubeadmin, which commands will also be transformed into the REST calls format.

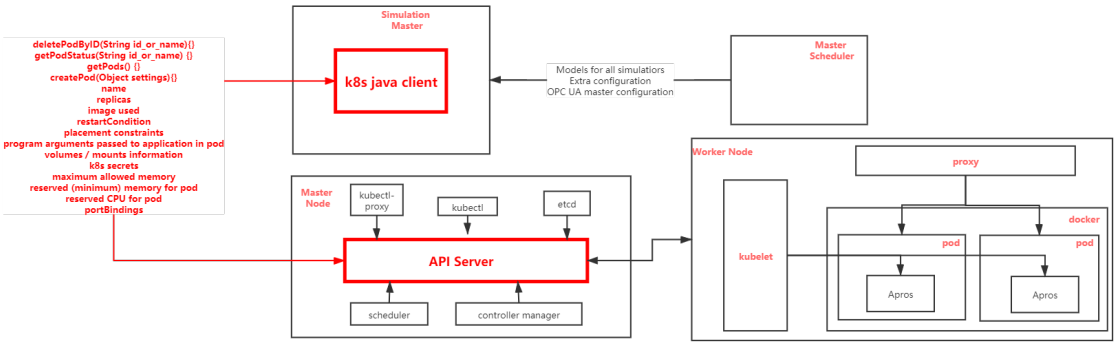

Figure 19: The architecture of the whole testing platform.

Moreover, users can also interact with the Kubernetes cluster directly with the REST calls. The second method can be implemented by utilizing the Kubernetes Java libraries. These libraries help developers write Java applications using the Kubernetes REST API and do not need to implement their libraries or REST API calls. There are many Kubernetes API client libraries in many popular languages, including Java, Go, and Python. The Kubernetes client library used in this experiment is the Kubernetes Java library.

As shown in figure 19, several Java methods are implemented by this thesis to interact with the Kubernetes cluster. For example, deletePodByID(String id_or_name) is used to delete the Pod specified with its pod ID. Figure 19 also presents the most critical method to operate the Kubernetes cluster, which is creating pods. This method createPod(Object settings) is used to create pods according to the object (settings). This object provides all the specifications required to create a container or a pod, such as an image name, port number, and volumes. Based on this Java application, the simulator containers can be deployed as pods in the Kubernetes cluster so that the co-simulation platform can also be working in Kubernetes.

5 Testing and Evaluation

The experiment results and the performance evaluation of the containerized simulators will be introduced in this chapter. Section 5.1 presents two testing modes: 500 testing signals mode, and control and process mode. Section 5.2 gives the OPC UA configuration files generated by the implemented Java application. Section 5.3 proves the Kubernetes Java client works by creating and deleting pods in the created Kubernetes cluster via Java methods. Section 5.4 tests the implemented Apros Dockerfile, image, and container, thus ensuring the Dockerfile is created correctly for the simulation software. Section 5.5 is the most important section where the results of co-simulation tests are presented. Section 5.6 discusses the future directions for this thesis.

5.1 Testing Cases

5.1.1 500 Testing Signals

The basic idea of the test is that 500 signal items and loopback setpoints will be generated by SCL scripts for two Apros containers, and let these two Apros containers exchange the 500 items data through OPC UA connections.
These 500 signals and loopback setpoints constitute the first test case where the generated signals in Apros1 will be sent to loopback setpoints in Apros2 and then send the signals back to Apros1. Based on this basic testing unit, 500 same testing units will be established for both Apros1 and Apros2.
The only method to test these 500 testing cases in Apros is to configure the OPC UA connections. Specifically, Apros1 will be configured as the UA server, and the Apros2 will be configured as the UA client via ServerConfig.xml file for each of the Apros instance. Moreover, after the UA server and client configuration, the specific information on connecting item pairs should be configured and treated carefully.
Because it is impossible to make the ConnectionConfig.xml file with hundreds or thousands of testing cases manually, the XML file generator implemented in Java will be used to firstly read the read from the SCL scripts and then generate the ConnectionConfig.xml file automatically. This XML file generator is significant because it successfully solved one critical problem: it is challenging to configure the ConnectionConfig.xml file manually, especially when there is a significant number of testing cases need to be configured via pairs. Figure 20 presents the diagrams of two items used in two Apros models to communicate via an OPC UA connection.

5.1.2 Control Model and Process Model

Two models are separately used in two different Apros containers in Docker Swarm. These two models are the control model shown in figure 22, and the process model shown in figure 23. For each of these two Apros models, five items are waiting to be connected to the other five items in another Apros model. Therefore, there will be five OPC UA connections created according to these five-item pairs in the control model and process model.

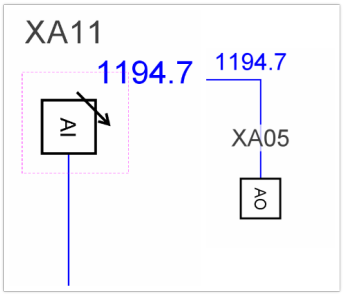

Figure 20: The item pair of one OPC UA connection.

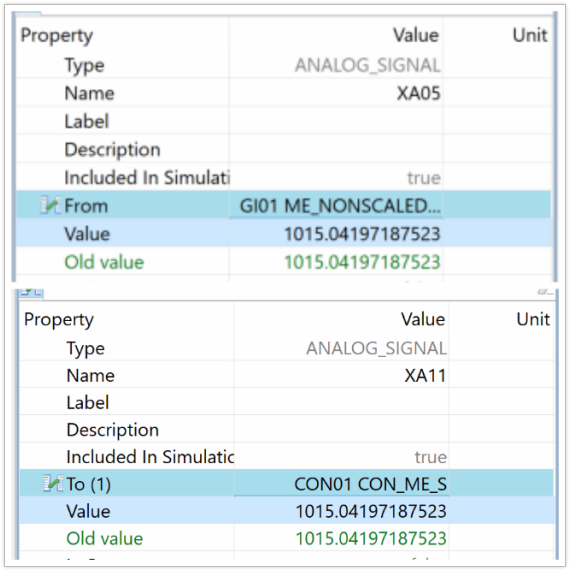

Figure 21: The comparison of XA11 and XA05 in two Apros instances.

5.2 Testing the OPC UA Configuration Files

The second testing case mentioned in section 5.1 will be used to test the manually created OPC UA XML files in two Apros instances. These two Apros instances are named as Apros1 and Apros2. Apros1 is used to run the control model, while the Apros2 is used to run the process model. Moreover, there are five items in each of these Apros instances, and each item is connected to the corresponded item in another Apros instance via the OPC UA connections. The testing scenario in the experiment is that the item XA11 in the Apros1 and XA05 in the Apros2 will be connected via the OPC UA interface. The expected result in this experiment is that the data of these two items should be the same. The value of the item can be checked in the Apros, as shown in figure 21. From the results of these two items, the value of XA11 and the value of XA05 are the same, which means these two items are connected via OPC UA connection, and the data of these two items can be transmitted in real-time.

Table 1: The testing results for five methods in the Kubernetes Java client

Methods \ Times	1	2	3	4	5	6	7	8	9	10	Average
APIConnection()	3.302s	3.244s	3.319s	3.273s	3.476s	3.331s	3.431s	3.353s	3.633s	3.327s	3.367s
getPods()	0.232s	0.225s	0.227s	0.239s	0.236s	0.225s	0.224s	0.235s	0.232s	0.229s	0.230s
createPod()	0.211s	0.212s	0.211s	0.213s	0.221s	0.207s	0.212s	0.223s	0.215s	0.212s	0.214s
deletePodByID()	0.231s	0.233s	0.275s	0.283s	0.235s	0.281s	0.286s	0.295s	0.230s	0.233s	0.257s
getPodStatus()	0.200s	0.202s	0.201s	0.199s	0.201s	0.199s	0.208s	0.202s	0.196s	0.196s	0.200s

5.3 Testing the Kubernetes Java Client

The Kubernetes Java client is used to interact with the Kubernetes cluster via API server. There are several implemented libraries for many popular languages, e.g., Go, Java, and Python, provided by Kubernetes so that developers can use these libraries to develop complex and flexible applications based on Kubernetes. The Kubernetes library used in this experiment is the Java library. This Kubernetes Java library provides all methods used to interact with the Kubernetes cluster. For example, many methods, such as creating a pod, getting the list of pods, and deleting the pod by its ID, are used to implement a more complex application. This Java application is one of the most critical implementations in this thesis because the Kubernetes Java client helps develop container-based applications more scalable.

The experiment implemented in this section is that five methods of this Java client are tested ten times and calculate the duration time for each of the tests. The average duration time is also given for each of the methods. All the testing results for the execution time can be found in the table 1. The testing results present the average execution time (3.367s, 0.230s, 0.214s, 0.257, and 0.200s) for these five methods. Even though the duration for connecting to the Kubernetes API server is 3.367s averagely, the method of APIConnection() only needs to be executed once before other methods. Once connecting this Java client to the target Kubernetes API server, the other four methods can be used without any more client-server connection. Except for this server connection time, the other four methods can be executed with short average execution time.

The testing results prove that the scalability can be achieved for running multiple containerized simulators by comparing it with running standalone simulator instances. The first reason is the container technology. The container virtualization enables deploying applications fast and scalable by indicating the number of replicas. The second reason is the Kubernetes Java client. By utilizing the Kubernetes libraries of preferred language, all the commands used to deploy simulators in the Kubernetes cluster can be implemented by the client and be executed by the Kubernetes API server automatically. By comparing with manually deploying multiple simulators, automatically distributing containerized simulators by designed Java applications achieves better performance with a better scalability capability.

5.4 Testing Two Apros Containers without OPC UA

As designed in this thesis, the objective is that the Apros should be containerized via Docker, and different Apros containers should cooperate to have a better performance. Therefore, the first step to test the Apros containers is to make a Dockerfile, which should be used to build the Apros image.
The Dockerfile in Appendix A presents all the steps to build the Apros image, including indicating the working directory, default entrypoint command, and volume. After configuring the Dockerfile, the Apros image can be built from it. Using the Docker command in Appendix E, the Apros container can be created from the designed Apros Dockerfile, and the related containers can be created in the Docker swarm cluster. The testing result proves that the Dockerfile was created correctly, and the Docker swarm cluster is ready to support the Apros containers meeting all runtime requirements.

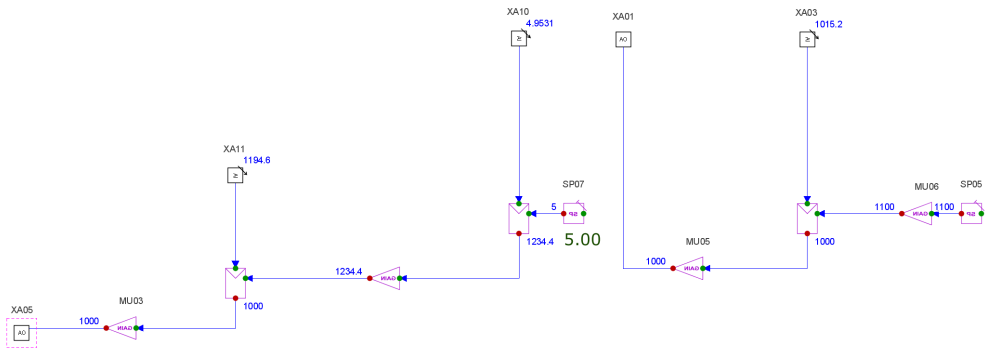

Figure 22: The diagram of control model.

5.5 Testing Two Apros Containers via OPC UA Interfaces

To run the multiple Apros containers in Docker Swarm, several steps need to be implemented and tested under different scenarios. The first step is to test the headless-version Apros instance that can work by executing the SCL commands without the Apros UI, so that make sure the containerized headless-version Apros can have the same performance as the non-containerized one. The second step is to test the Apros image built in the last section so that the Apros containers can perform well in Docker swarm. This step ensures that the necessary runtime, dependencies, and binary files are added into the Apros image properly. After testing the main components of the containerized Apros, the OPC interfaces, e.g., OPC UA client and server, and Apros items in OPC UA connections need to be tested so that the

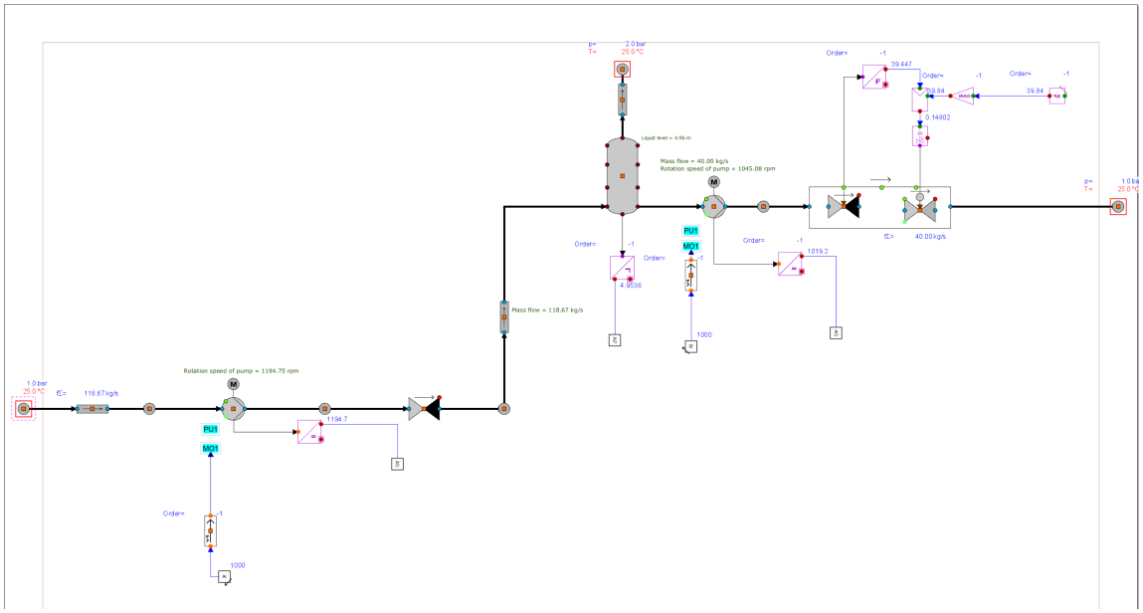

Figure 23: The diagram of the process model.

UAExpert running on the host can find and connect to the OPC UA servers in the Apros containers, and display all items defined in the OPC UA connections.
Subsection 5.3.1 presents the testing results for non-containerized headless-version Apros instances, and subsection 5.3.2 presents the testing results of Apros containers.

5.5.1 Testing Two Headless-version Apros Instances

Because the headless-version Apros will be containerized in Docker and be used as Apros containers, the headless-version Apros binaries need to be tested firstly according to the main functionalities, e.g., modeling and simulation, and the OPC UA interfaces components. Furthermore, it is challenging to troubleshoot inside containers if the functionality components or OPC UA components do not work correctly. Therefore, the first step is to test the headless-version Apros instances with testing SCL scripts and two testing models. The testing SCL scripts indicate some commands to be executed in Apros instances, such as creating specific diagrams, setting the experiment speed, and other commands used to control the whole experiment process. Besides the testing SCL scripts used in this testing, two testing models are separately used in two testing Apros instances. These two testing models are the control model and process model, and each of them has five items used to communicate with five items in other models.
Figure 24 presents the testing architecture of two Apros instances and UAExpert running on the host machine. Two Apros instances and UAExpert are running as two independent processes on the host machine. These two Apros instances are connected via the OPC UA interfaces. For each OPC UA interface, one OPC UA server will be exposed to connect to other OPC UA clients, such as the OPC UA

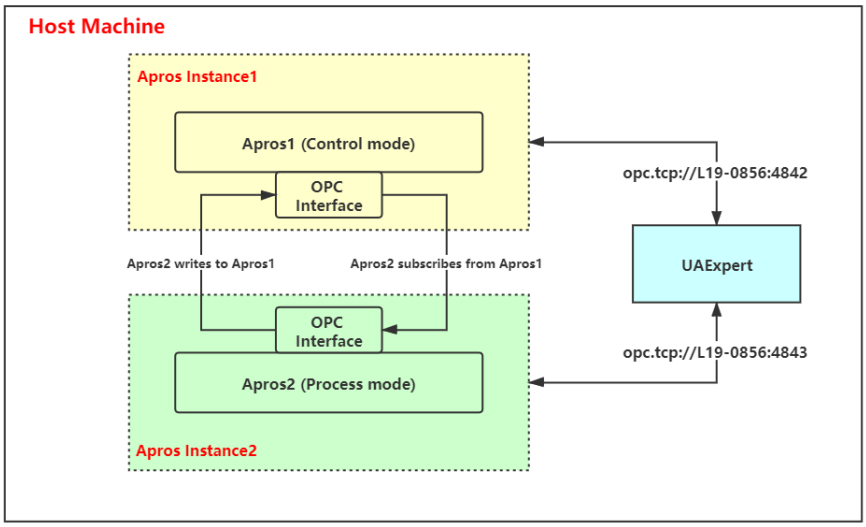

Figure 24: The architecture of the testing platform for two Apros instances.

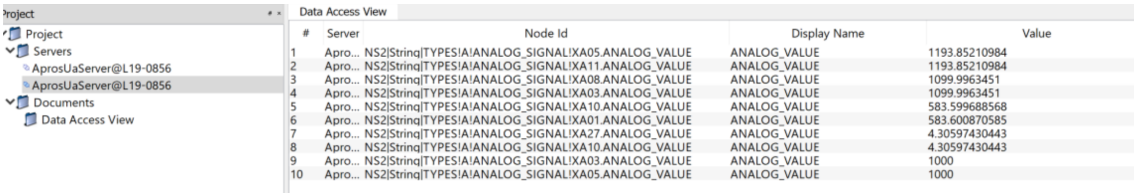

Figure 25: The testing results of two Apros instances viewed in UAExpert.

client enabled in other Apros instances and the UAExpert. The OPC UA servers use customized OPC UA Url to expose. For example, the OPC Url of the Apros1 is opc.tcp://L19-0856:4842. This OPC Url can be found by both the Apros2 and the UAExpert so that the OPC UA clients can connect to this OPC UA server.
The UAExpert connects to the Apros1 and Apros2 via OPC UA server-client architecture. Then all the items used in the OPC UA connections can be viewed in the UAExpert, for example, the XA05 ANALOG VALUE in Apros2 and XA11 ANALOG VALUE in Apros1, as shown in figure 25. Figure 25 presents the data subscribed from Apros1 and Apros2, and the data changes in real-time via OPC UA connections. Five item pairs are defined in Apros1 and Apros2, and these five-item pairs communicate via OPC UA connections. Therefore, the testing results should be that the five-item pairs are the same, approximately with each other, and the data viewed in the UAExpert should be updated in real-time. From the results collected by UAExpert shown in figure 25, the experiment for two Apros instances running on host machine communicating via OPC UA is successful, and the results collected prove the headless-version Apros is working by executing testing SCL commands. Moreover, because the five-item pairs data are the same approximately, it proves that the OPC UA interfaces of Apros are working and the configuration files, e.g., ConnectionConfig.xml and ServerConfig.xml, are created correctly.

The experiment in subsection 5.3.1 provides the experiment results proving the headless-version Apros with SCL testing scripts and OPC UA configuration XML files are working as expected. Based on this result, the later subsection 5.3.2 will do a similar experiment but in the Docker swarm mode where the containerized Apros is used rather than the headless-version Apros instances running on the host machine.

5.5.2 Testing two Apros containers in Docker Swarm

After proving the headless-version Apros can work well by executing the SCL commands, the possibility of running the Apros containers based on the headless-version Apros is considered the main objective of this subsection. Correctly, instead of running multiple Apros instances on the host machine, two containerized Apros will be running in the Docker Swarm, and all the required OPC UA configuration XML files will be created and mounted in Apros containers. The UAExpert will still run on the host machine but with different OPC UA Url to connect. As shown in figure 26, the IP addresses of two Apros containers are 172.21.252.211 and 172.21.251.140, respectively. Therefore, the two URLs used by UAExpert are set as opc.tcp://172.21.252.211:4842 and opc.tcp://172.21.251.140:4843. The IP address and port number for each Apros container should be explicitly defined in the OPC UA configuration XML files so that OPC UA connections can be created and other OPC UA client, e.g., UAExpert, can connect to these two OPC UA servers successfully. Figure 26 gives the general architecture of the experiment used in this subsection.

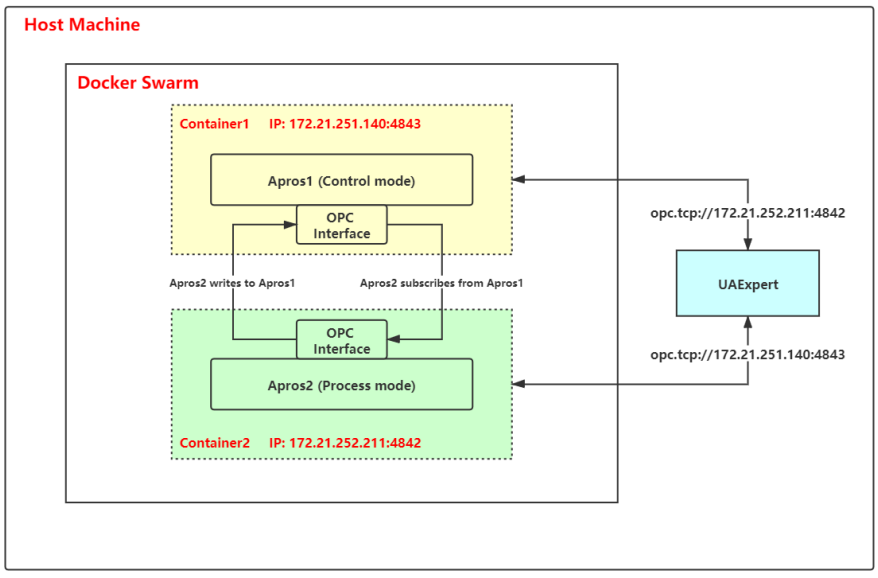

Figure 26: The architecture of the testing platform for two Apros containers.

As shown in figure 26, the UAExpert is used to connect to the OPC UA interfaces of Apros1 and Apros2 containers. The same item pairs used to communicate are the same as used in the subsection 5.3.1, and the only difference for the UAExpert is the OPC UA URLs. The two OPC UA URLs used in this experiment are defined

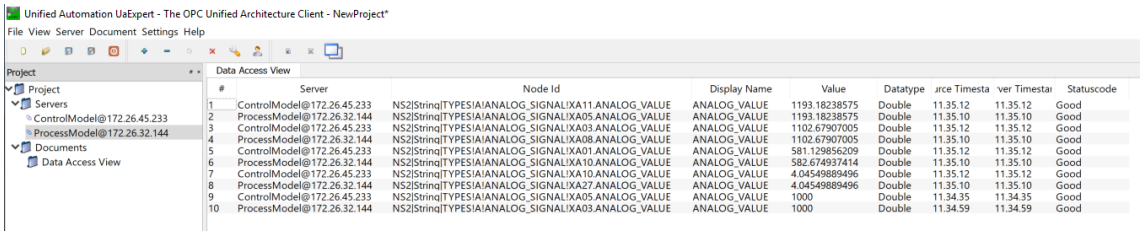

Figure 27: The testing results of two Apros containers viewed in the UAExpert.

according to the two Apros containers' IP address. Figure 27 presents the collected results of 5 item pairs. As shown in figure 27, each pair of these five pairs has the same data for two items in the same pair.
The results collected and viewed in the UAExpert can be used to validate two factors for this experiment. The first factor is that the Apros images are created correctly, and Apros containers can work correctly as designed. Moreover, the second factor can be proved the correctness of OPC UA configuration XML files, which are mainly responsible for creating the OPC UA servers and creating OPC UA connections with other OPC UA servers.

5.6 Future Work

This thesis implemented most parts of the designed experiment. However, there are still components and tests that have not been done yet. In detail, the testing case of two models used in two Apros containers proves that the OPC UA configuration files and Apros Dockerfile are working as expected in the Docker swarm cluster. Moreover, the Kubernetes Java client test presents the result that this Java client interacts with the Kubernetes API server successfully with designed methods, such as creating and deleting pods in the cluster. However, the large scale testing case with 500 signals has not been tested yet in the co-simulation platform due to the high possibility of crashes in Apros. Specifically, the Apros process will crash when the number of items used in OPC UA connections is huge. The reason for this type of OPC UA crashes is not clear yet. The plan of future implementation is described as follows:

- OPC UA configuration files will be designed and optimized so that more items in the Apros containers can be selected to transmit data in the OPC UA connections. Because the OPC UA configuration method in Apros is not very interactive and efficient, it is challenging to configure the OPC UA XML files correctly according to a significant number of selected items. Thus the existing configuration XML files do not work as expected with these 500 testing signals.

- Even though these OPC UA configuration XML files and Kubernetes API calls can be generated automatically via Java client, the method of deploying Apros containers in the Docker swarm is not totally automatic and efficient due to the dynamic IP addresses of containers and OPC UA configuration files. Multiple containers created via the Docker swarm service will be allocated with dynamic IP addresses. This feature of the dynamic IP address in the

Docker swarm mode makes it impossible to configure the OPC UA XML files before running this Apros Docker swarm service, because the OPC XML files require the IP addresses of OPC UA server and client. Future work is to design and implement an elegant way of automatically deploying Apros containers via Docker swarm service so that the whole process of co-simulation could be totally automatic according to user requirements.

- By creating a Docker swarm cluster or Kubernetes cluster in the cloud platform, the cluster's architecture could be complicated for supporting advanced applications and services by users. For example, the nodes in the cluster could be distributed to different locations worldwide instead of being created in the same region. The latency issue has to be noticed if the nodes in the cluster are distributed because the transmission time between the master node and worker nodes could be significant and unacceptable. The plan is to implement a method to select the worker nodes or VMs according to their tags, such as region: EU and country: Finland. By selecting the proper tags of worker nodes, the VMs created in the cluster with the same tags will be selected to receive and execute the workloads with lower latency.

6 Conclusion

Many industry factories and technical systems are becoming more sophisticated, making these factories and systems challenging to be modeled and simulated by simulation software. This is why the co-simulation technologies and platforms have been developed and improved by many companies and experts in many industrial areas. Within a large and complex technical system or factory, the whole entity can be divided into several simpler sub-units that are easier to be modeled and simulated by multiple simulators. Moreover, co-simulation technologies can provide many advantages, such as computational efficiency, more accurate results, and flexibility for modeling and simulation.

It is noticed that running applications in Docker containers are the trend in the IT world. Docker container technology is considered as one of the fastest developing and growing technologies in the recent history of the IT world. The container virtualization tool, Docker, enables users to build customized images, pull required images built officially or privately, and distribute containers to available nodes in the created cluster. Therefore, Docker container virtualization technology can enhance developing, testing, and distributing applications and software. Specifically, Docker container technology provides the container platform where multiple containers can be launched quickly with either Windows base images or Linux base images. This advantage makes deploying simulation software containers easier and more convenient than creating multiple VMs in the cloud platform because running cooperated simulators usually requires low latency, high scalability, and flexibility.

Based on the advantages of cooperative simulations and container virtualization technologies, creating the co-simulation platform and running cooperative containerized simulators with Docker container technology can achieve better performance than running a single complex simulation process on the host machine. The co-simulation platform implemented in this thesis provides the platform with some functions to run multiple cooperated simulators efficiently and automatically. These functions can automatically receive simulation initialization model and scripts, generate required XML files for OPC UA connections, and launch simulation containers in Docker swarm or Kubernetes via REST API calls. Specifically, if a user wants to optimize the existing system or wants to know what is the best option for one component in the system, the user can pass the initial models and scripts of testing to this platform, and then the data received from the user will be processed by the co-simulation platform. All configurations and settings to run multiple cooperative containerized simulators will be prepared by this co-simulation platform so that multiple simulator containers can work cooperatively in the Docker cluster and Kubernetes cluster.

References

[1] European Commission (EC). Europe 2020: A strategy for smart, sustainable and inclusive growth. *Working paper {COM (2010) 2020}*, 2010.

[2] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[3] Zhou Ji. Intelligent manufacturing—main direction of "made in china 2025". *China Mechanical Engineering*, 26(17):2273–2284, 2015.

[4] JP Holdren, T Power, G Tassey, A Ratcliff, and L Christodoulou. A national strategic plan for advanced manufacturing. *US National Science and Technology Council, Washington, DC*, 2012.

[5] Baotong Chen, Jiafu Wan, Lei Shu, Peng Li, Mithun Mukherjee, and Boxing Yin. Smart factory of industry 4.0: Key technologies, application case, and challenges. *IEEE Access*, 6:6505–6519, 2017.

[6] Michael Schluse, Marc Priggemeyer, Linus Atorf, and Juergen Rossmann. Experimentable digital twins—streamlining simulation-based systems engineering for industry 4.0. *IEEE Transactions on Industrial Informatics*, 14(4):1722–1731, 2018.

[7] Darko Androcec, Neven Vrcek, and Jurica Seva. Cloud computing ontologies: A systematic review. In *Proceedings of the third international conference on models and ontology-based design of protocols, architectures and services*, pages 9–14, 2012.

[8] Tuomas Miettinen et al. Synchronized cooperative simulation: Opc ua based approach. 2012.

[9] Edward N Zalta, Uri Nodelman, Colin Allen, and John Perry. Stanford encyclopedia of philosophy, 1995.

[10] Hossein Arsham. Systems simulation: The shortest route to applications. *National Science Foundation*, 1995.

[11] Kerry A Dennis and Don Harris. Computer-based simulation as an adjunct to ab initio flight training. *The International Journal of Aviation Psychology*, 8(3):261–276, 1998.

[12] Borja Rodríguez, Francisco González, Miguel Ángel Naya, and Javier Cuadrado. Assessment of methods for the real-time simulation of electronic and thermal circuits. *Energies*, 13(6):1354, 2020.

[13] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. Co-simulation: State of the art. *arXiv preprint arXiv:1702.00686*, 2017.

[14] Apros. Apros in brief. http://www.apros.fi/en/apros_in_brief_2.

[15] Apros. Training and education. http://www.apros.fi/en/applications/training_and_education.

[16] Sebastian Lehnhoff, Sebastian Rohjans, Mathias Uslar, and Wolfgang Mahnke. Opc unified architecture: A service-oriented architecture for smart grids. In *2012 First International Workshop on Software Engineering Challenges for the Smart Grid (SE-SmartGrids)*, pages 1–7. IEEE, 2012.

[17] Ioan Ungurean, Nicoleta-Cristina Gaitan, and Vasile Gheorghita Gaitan. An iot architecture for things from industrial environment. In *2014 10th International Conference on Communications (COMM)*, pages 1–4. IEEE, 2014.

[18] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC unified architecture*. Springer Science & Business Media, 2009.

[19] Stefan-Helmut Leitner and Wolfgang Mahnke. Opc ua–service-oriented architecture for industrial applications. *ABB Corporate Research Center*, 48:61–66, 2006.

[20] G Martinov, A Issa, and L Martinova. Controlling can servo step drives and their remote monitoring by using protocol opc ua. In *2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pages 1–5. IEEE, 2019.

[21] Sten Grüner, Julius Pfrommer, and Florian Palm. Restful industrial communication with opc ua. *IEEE Transactions on Industrial Informatics*, 12(5):1832–1841, 2016.

[22] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[23] Charles Anderson. Docker [software engineering]. *Ieee Software*, 32(3):102–c3, 2015.

[24] Sunil Mohanty et al. Evaluation of serverless computing frameworks based on kubernetes. 2018.

[25] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[26] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, pages 275–287, 2007.

[27] EN Preeth, Fr Jaison Paul Mulerickal, Biju Paul, and Yedhu Sastri. Evaluation of docker containers based on hardware utilization. In *2015 International Conference on Control Communication & Computing India (ICCC)*, pages 697–700. IEEE, 2015.

[28] Docker Documents. Containers and virtual machines. https://https://docs.docker.com/get-started/#containers-and-virtual-machines.

[29] Docker Documents. About storage drivers. https://docs.Docker.com/storage/storagedriver/.

[30] David Jaramillo, Duy V Nguyen, and Robert Smart. Leveraging microservices architecture by using docker technology. In *SoutheastCon 2016*, pages 1–5. IEEE, 2016.

[31] Joao Rufino, Muhammad Alam, Joaquim Ferreira, Abdur Rehman, and Kim Fung Tsang. Orchestration of containerized microservices for iiot using docker. In *2017 IEEE International Conference on Industrial Technology (ICIT)*, pages 1532–1536. IEEE, 2017.

[32] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[33] Docker Documents. How nodes work. https://docs.Docker.com/engine/swarm/how-swarm-mode-works/nodes/.

[34] Docker Documents. How swarm works. https://docs.Docker.com/engine/swarm/howswarm-mode-works/services/.

[35] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2658–2659. IEEE, 2017.

[36] Kubernetes Documents. What is kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

A Dockerfile Used to Build the Apros Image

This appendix presents the Dockerfile used in the Docker to build the Apros image. This Dockerfile was created and maintained by Miro Eklund.

```
FROM mcr.microsoft.com/windows/servercore:ltsc2016
MAINTAINER Miro Eklund
RUN mkdir "c:/workdir"
COPY ["simulator/gzip.exe", "c:/workdir/gzip.exe"]
COPY ["7-Zip", "C:/7-Zip"]
COPY ["server/CodeMeterRuntimeReduced.exe", "C:/
   CodeMeterInstallation.exe"]
RUN call "C:/CodeMeterInstallation.exe" /ComponentArgs
   "*":"/qn"
RUN mkdir "c:/model" && \
    mkdir "c:/workdir/output" && \
    mkdir "c:/script"
CMD [""]
RUN mkdir "c:/license"
COPY ["batchfiles", "c:/batchfiles"]
ENTRYPOINT ["c:/batchfiles/Apros_entrypoint.bat"]
COPY ["simulator/TS3AprosHeadlessThermal", "c:/Simulator
   "]
RUN ren C:\simulator\AprosHeadless.ini eclipsec.ini && \
del "C:\simulator\AprosHeadless.exe"
WORKDIR "c:/workdir"
```

B The Output of Creating A Docker and Kubernetes Cluster

Figure B1: The output of creating a Docker cluster.

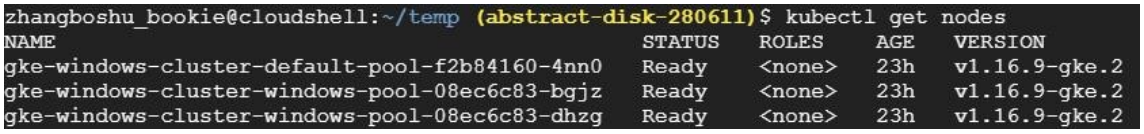

Figure B2: The output of creating a Kubernetes cluster.

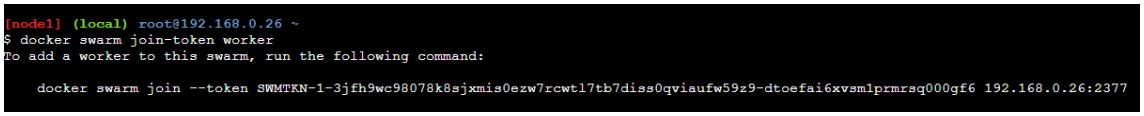

Figure B3: The command and output of obtaining the join-token.

C ConnectionConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<OpcServerConfig xmlns:xsi="http://www.w3.org/2001/
   XMLSchema" xmlns="http://www.vtt.fi/OPCUA/
   connectionconfig.xsd">
<DxConnectionConfig>
  <Tick>
    <Length>0.1</Length>
  </Tick>
  <Synchronization>false</Synchronization>
  <ServerConnections>
    <Connection>
      <Enabled>true</Enabled>
      <SubscribeeUsesWrite>false</SubscribeeUsesWrite>
      <Subscriber>
        <Url>opc.tcp://L19-0856:4842</Url>
        <Name>Process Server</Name>
      </Subscriber>
      <Subscribee>
        <Url>opc.tcp://L19-0856:4843</Url>
        <Name>Control Server</Name>
      </Subscribee>
      <ItemConnections>
        <DxConnection
                        DxItemId="DX.DXConnectionsRoot.
                           Control␣Server.toProcess"
                        DxItemName="from␣Control.XA05"
                        TargetItemPath=""
                        TargetItemName=""
                        TargetItemId="TYPES!A!
                           ANALOG_SIGNAL!XA03.
                           ANALOG_VALUE"
            TargetNamespaceIndex="2"
                        TargetItemDataType="Double"
                        SourceItemPath=""
                        SourceItemName=""
                        SourceNamespaceIndex="2"
                        SourceItemId="TYPES!A!
                           ANALOG_SIGNAL!XA05.
                           ANALOG_VALUE"
                        Scale="1.0"
                >
          <BrowsePath>Control Server</BrowsePath>
          <Description/>
```

```
      <DefaultOverrideValue/>
      <SubstituteValue/>
      <VendorData/>
        </DxConnection>
    <DxConnection
                    DxItemId="DX.DXConnectionsRoot.
                       Control␣Server.toProcess"
                    DxItemName="from␣Control.XA01"
                    TargetItemPath=""
                    TargetItemName=""
                    TargetItemId="TYPES!A!
                       ANALOG_SIGNAL!XA10.
                       ANALOG_VALUE"
        TargetNamespaceIndex="2"
                    TargetItemDataType="Double"
                    SourceItemPath=""
                    SourceItemName=""
                    SourceNamespaceIndex="2"
                    SourceItemId="TYPES!A!
                       ANALOG_SIGNAL!XA01.
                       ANALOG_VALUE"
                    Scale="1.0"
            >
      <BrowsePath>Control Server</BrowsePath>
      <Description/>
      <DefaultOverrideValue/>
      <SubstituteValue/>
      <VendorData/>
        </DxConnection>
  </ItemConnections>
</Connection>
    <Connection>
  <Enabled>true</Enabled>
  <SubscribeeUsesWrite>true</SubscribeeUsesWrite>
  <Subscriber>
    <Url>opc.tcp://L19-0856:4843</Url>
    <Name>Control Server</Name>
  </Subscriber>
  <Subscribee>
    <Url>opc.tcp://L19-0856:4842</Url>
    <Name>Process Server</Name>
  </Subscribee>
  <ItemConnections>
    <DxConnection
                    DxItemId="DX.DXConnectionsRoot.
```

```
                    Process␣Server.toControl"
                DxItemName="from␣Process.XA05"
                TargetItemPath=""
                TargetItemName=""
                TargetItemId="TYPES!A!
                    ANALOG_SIGNAL!XA11.
                    ANALOG_VALUE"
    TargetNamespaceIndex="2"
                TargetItemDataType="Double"
                SourceItemPath=""
                SourceItemName=""
                SourceNamespaceIndex="2"
                SourceItemId="TYPES!A!
                    ANALOG_SIGNAL!XA05.
                    ANALOG_VALUE"
                Scale="1.0"
        >
  <BrowsePath>Process Server</BrowsePath>
  <Description/>
  <DefaultOverrideValue/>
  <SubstituteValue/>
  <VendorData/>
    </DxConnection>
<DxConnection
                DxItemId="DX.DXConnectionsRoot.
                    Process␣Server.toControl"
                DxItemName="from␣Process.XA27"
                TargetItemPath=""
                TargetItemName=""
                TargetItemId="TYPES!A!
                    ANALOG_SIGNAL!XA10.
                    ANALOG_VALUE"
    TargetNamespaceIndex="2"
                TargetItemDataType="Double"
                SourceItemPath=""
                SourceItemName=""
                SourceNamespaceIndex="2"
                SourceItemId="TYPES!A!
                    ANALOG_SIGNAL!XA27.
                    ANALOG_VALUE"
                Scale="1.0"
        >
  <BrowsePath>Process Server</BrowsePath>
  <Description/>
  <DefaultOverrideValue/>
```

```
          <SubstituteValue/>
          <VendorData/>
            </DxConnection>
        <DxConnection
                        DxItemId="DX.DXConnectionsRoot.
                           Process␣Server.toControl"
                        DxItemName="from␣Process.XA08"
                        TargetItemPath=""
                        TargetItemName=""
                        TargetItemId="TYPES!A!
                           ANALOG_SIGNAL!XA03.
                           ANALOG_VALUE"
          TargetNamespaceIndex="2"
                        TargetItemDataType="Double"
                        SourceItemPath=""
                        SourceItemName=""
                        SourceNamespaceIndex="2"
                        SourceItemId="TYPES!A!
                           ANALOG_SIGNAL!XA08.
                           ANALOG_VALUE"
                        Scale="1.0"
                >
          <BrowsePath>Process Server</BrowsePath>
          <Description/>
          <DefaultOverrideValue/>
          <SubstituteValue/>
          <VendorData/>
            </DxConnection>
      </ItemConnections>
    </Connection>
  </ServerConnections>
</DxConnectionConfig>
<General>
</General>
</OpcServerConfig>
```

D SCL Scripts Used to Generate Apros Models and Start the Simulation

(1) generators.scl:

```
import "Apros/All"
import "Simantics/ProceduralUserComponent"
import "http://www.Apros.fi/Combustion/Configuration-6.0"
    as CONF

createNSignals diagram n moduleType1 moduleType2 = do
    outputSignalName = "SINE_OUTPUT_SIGN"
    inputSignalName = "MULTIPLYER_INPUT_SIGN(1)"
    syncWrite \() -> forN n \k -> do
        x = 30 * fromInteger (k `mod` 10) + 20
        y = 30 * fromInteger (k `div` 10) + 20
        aaddSW diagram moduleType1 "SG\(k+1)" (x, y)
        x = 30 * fromInteger (k `mod` 10) + 30
        y = 30 * fromInteger (k `div` 10) + 30
        aaddSW diagram moduleType2 "MUL\(k+1)" (x, y)
    syncGraph ()
    forN n \k -> do
        aconnectExternalSignal "SG\(k+1)"
           outputSignalName "XAO\(k+1)"
        asetFlagType "XAO\(k+1)" "IO"
        aconnectExternalSignal "MUL\(k+1)"
           inputSignalName "XAI\(k+1)"
        asetFlagType "XAI\(k+1)" "IO"

n = 500
diagramName = "Signals"
moduleType1 = "SINE_WAVE"
moduleType2 = "MULTIPLYER"
diagram = createAutomationDiagram (configurationOf
   currentModel) diagramName
createNSignals diagram n moduleType1 moduleType2
```

(2) setpoints.scl:

```
import "Apros/All"

diagramName = "LoopbackSetpoints"
moduleType = "SET_POINT"
n = 500

diagram = createAutomationDiagram (configurationOf
   currentModel) diagramName
```

```
syncWrite \() -> forN n \k -> do
    x = 20 * fromInteger (k `mod` 10) + 20
    y = 10 * fromInteger (k `div` 10) + 20
    aaddSW diagram moduleType "SP\(k+1)" (x, y)
```

(3) test.scl:

```
include "Apros/Sequences" as AprosSequences
import "Simantics/Sequences" as SSequences
include "Apros/All"
include "Files" as Files
include "StringIO"
include "Simantics/DB" as SimanticsDB
include "Apros/Experiment" as AprosExperiment

loadModel icName = do
    forceDumpICs = False
    experimentName = "Experiment"
    testScriptFileName = "test.scl"
    modelName = "model"
    m = if(forceDumpICs) then importModelDump (file
       modelName) True else importModel (file modelName)
    match(syncRead (\_ -> possibleResource icName)) with
        Just ic -> do
            loadInitialCondition (ic :: InitialCondition)
        Nothing -> do
            modelName = syncRead(\_ ->  (SimanticsDB.
               uriOf (m :: Resource)))
            icURI = modelName + "/" + icName
            loadInitialCondition $ (syncRead (\_ ->
               resource icURI) :: InitialCondition)
    ()

    syncActivateModel m
    syncWrite $ \_ -> AprosExperiment.activateExperiment
       $ child m experimentName
    ()

main :: Boolean -> <Proc > ()
main dummy = do
    loadModel "Initial%20Condition"
    setSpeed 1.0
    result = syncWrite (\() -> newResource ())
    step 600.0
```

E The Docker Command Used to Run Apros Service in the Docker Swarm

```
$ docker service create --dns 130.233.251.4 --constraint
   node.platform.OS==windows --mount type=bind,source=C:\
   Users\lyut1\license,destination=c:/license,readonly=
   true --name Aprosheadless --replicas 1 --restart-
   condition=none --limit-memory 3221225472 --reserve-
   memory 3221225472 --secret src=reposimupediacom,target
   ="c:/credentials.txt" Apros-6-thermal:6.09.33
   unique_run_id <url to model>/Thermal60933Model_Tuojian
   .zip <url to script>/Thermal60933Script_Tuojian.zip
   username 2g test.scl 1
```

F The Java Application Used to Generate ConnectionConfig.xml File

```
import Javax.xml.parsers.DocumentBuilder;
import Javax.xml.parsers.DocumentBuilderFactory;
import Javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;

import Java.io.File;
import Java.io.FileOutputStream;
import Java.io.IOException;
import Java.io.FileNotFoundException;

import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.XMLSerializer;

import Connection.Connection;
import SCL_script_Reader.SCLReader;

public class Main {

        public static void main(String[] args) throws
           ParserConfigurationException, FileNotFoundException {

                // read the scl file and get the number of units
                int num = 0;
                num = SCLReader.num_getter("setpoints.scl");

                // create the xml document object
                Document xmlDoc = xmlDocInitializer();

                // input your opc ua xml data there, including
                   number of items, server url,
                // name, etc.
                xmlDoc = nodesBuilder(xmlDoc, num);

                // generate the upc ua xml file at current
                   location
                output(xmlDoc);
        }

        public static Document xmlDocInitializer() throws
           ParserConfigurationException {
```

```
        DocumentBuilderFactory docFactory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder = docFactory.
            newDocumentBuilder();
        Document xmlDoc = docBuilder.newDocument();
        xmlDoc.setStrictErrorChecking(false);
        return xmlDoc;
}

public static Document nodesBuilder(Document xmlDoc, int
    num) throws FileNotFoundException {

        Element OpcServerConfig = xmlDoc.createElement("
            OpcServerConfig");
        Element DxConnectionConfig = xmlDoc.
            createElement("DxConnectionConfig");
        Element General = xmlDoc.createElement("General"
            );
        Element Tick = xmlDoc.createElement("Tick");
        Element Length = xmlDoc.createElement("Length");
        Text text = xmlDoc.createTextNode("");
        Element Synchronization = xmlDoc.createElement("
            Synchronization");
        Element ServerConnections = xmlDoc.createElement
            ("ServerConnections");

        OpcServerConfig.setAttribute("xmlns:xsi", "http
            ://www.w3.org/2001/XMLSchema");
        OpcServerConfig.setAttribute("xmlns", "http://
            www.vtt.fi/OPCUA/connectionconfig.xsd");

        text = xmlDoc.createTextNode("0.1");
        Length.appendChild(text);
        Tick.appendChild(Length);
        DxConnectionConfig.appendChild(Tick);
        text = xmlDoc.createTextNode("false");
        Synchronization.appendChild(text);
        DxConnectionConfig.appendChild(Synchronization);

        // input the number of Dxconnection items

        Connection connection = new Connection();
        connection.setTagName("Connection");
        connection.setEnabled("true");
        connection.setSubscribeeUsesWrite("false");
        connection.getSubscriber().setUrl("opc.tcp://L19
            -0856:4842");
        connection.getSubscriber().setName("Signals␣
```

```
    Server");
connection.getSubscribee().setUrl("opc.tcp://L19
    -0856:4843");
connection.getSubscribee().setName("
    LoopbackSetpoints␣Server");
connection.getDxConnection().setDxItemId("DX.
    DXConnectionsRoot.LoopbackSetpoints␣Server.
    toSignals");
connection.getDxConnection().setDxItemName("from
    ␣LoopbackSetpoints.SP*");
connection.getDxConnection().setTargetItemId("
    TYPES!A!ANALOG_SIGNAL!XAI*.ANALOG_VALUE");
connection.getDxConnection().setSourceItemId("
    TYPES!S!SET_POINT!SP*.SP_VALUE");
Element Connections = connection.builder(xmlDoc,
     num);
ServerConnections.appendChild(Connections);

connection.setTagName("Connection");
connection.setEnabled("true");
connection.setSubscribeeUsesWrite("true");
connection.getSubscriber().setUrl("opc.tcp://L19
    -0856:4843");
connection.getSubscriber().setName("
    LoopbackSetpoints␣Server");
connection.getSubscribee().setUrl("opc.tcp://L19
    -0856:4842");
connection.getSubscribee().setName("Signals␣
    Server");
connection.getDxConnection().setDxItemId("DX.
    DXConnectionsRoot.Signals␣Server.
    toLoopbackSetpoints");
connection.getDxConnection().setDxItemName("from
    ␣Signals.XAO*");
connection.getDxConnection().setTargetItemId("
    TYPES!S!SET_POINT!SP*.SP_VALUE");
connection.getDxConnection().setSourceItemId("
    TYPES!A!ANALOG_SIGNAL!XAO*.ANALOG_VALUE");
Connections = connection.builder(xmlDoc, num);
ServerConnections.appendChild(Connections);

DxConnectionConfig.appendChild(ServerConnections
    );
OpcServerConfig.appendChild(DxConnectionConfig);
OpcServerConfig.appendChild(General);

xmlDoc.appendChild(OpcServerConfig);
return xmlDoc;
```

```
        }

        public static void output(Document xmlDoc) throws
            FileNotFoundException {

                // set outputFormat
                OutputFormat outFormat = new OutputFormat(xmlDoc
                    );
                outFormat.setIndenting(true);

                // declare the file
                File xmlFile = new File("ConnectionConfig.xml");
                // declare the fileOutputStream
                FileOutputStream outStream = new
                    FileOutputStream(xmlFile);

                // XMLSerializer to serialize the xml data with
                XMLSerializer serializer = new XMLSerializer(
                    outStream, outFormat);
                // the specified outputformat
                try {
                        serializer.serialize(xmlDoc);
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

G The Java Application Used to Generate Apros Pods in the Kubernetes Cluster

```
import io.kubernetes.client.ApiException;
import io.kubernetes.client.apis.ExtensionsV1beta1Api;
import io.kubernetes.client.custom.Quantity;
import io.kubernetes.client.models.ExtensionsV1beta1Deployment;
import io.kubernetes.client.models.
    ExtensionsV1beta1DeploymentSpec;
import io.kubernetes.client.models.
    ExtensionsV1beta1DeploymentStatus;
import io.kubernetes.client.models.
    ExtensionsV1beta1DeploymentStrategy;
import io.kubernetes.client.models.V1Container;
import io.kubernetes.client.models.V1ContainerPort;
import io.kubernetes.client.models.V1LabelSelector;
import io.kubernetes.client.models.V1ObjectMeta;
import io.kubernetes.client.models.V1PodSpec;
import io.kubernetes.client.models.V1PodTemplateSpec;
import io.kubernetes.client.models.V1ResourceRequirements;
import io.kubernetes.client.util.ClientBuilder;
import Java.io.IOException;
import Java.util.ArrayList;
import Java.util.HashMap;
import Java.util.List;
import Java.util.Map;

public class podCreaterFromPara {

        static ExtensionsV1beta1Deployment body = new
            ExtensionsV1beta1Deployment();
        static V1ObjectMeta metadata = new V1ObjectMeta();
        static ExtensionsV1beta1DeploymentSpec deploymentSpec =
            new ExtensionsV1beta1DeploymentSpec();
        static Map<String, String> Labels = new HashMap<String,
            String>();
        static V1LabelSelector selector = new V1LabelSelector();
        static ExtensionsV1beta1DeploymentStrategy strategy =
            new ExtensionsV1beta1DeploymentStrategy();
        static V1ObjectMeta metadata1 = new V1ObjectMeta();
        static ExtensionsV1beta1DeploymentStatus status = new
            ExtensionsV1beta1DeploymentStatus();
        static V1PodTemplateSpec podTemplateSpec = new
            V1PodTemplateSpec();
        static V1ContainerPort port = new V1ContainerPort();
        static V1PodSpec podSpec = new V1PodSpec();
        static List<V1Container> containers = new ArrayList<
```

```
    V1Container>();
static V1Container container = new V1Container();
static V1ResourceRequirements resources = new
    V1ResourceRequirements();
static List<V1ContainerPort> ports = new ArrayList<
    V1ContainerPort>();
static Map<String, Quantity> requests = new HashMap<
    String, Quantity>();

public static void main(String[] args) {

        Settings settings = new Settings();

        settings.getLabels().put("run", "hello");
        settings.getRequests().put("cpu", "100m");
        settings.getRequests().put("memory", "100Mi");
        settings.setImage("hello-node:v1");
        settings.setName("test5");
        settings.setPort(8080);


        try {
                createPod(settings);
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}

public static void createPod(Settings settings) throws
    IOException {

        ExtensionsV1beta1Api api = new
            ExtensionsV1beta1Api(ClientBuilder.standard()
            .build());

        readDataFromSettings(settings);
        writeDataToBody();

        // create a deployment
        try {
                ExtensionsV1beta1Deployment deploy = api
                    .createNamespacedDeployment("default"
                    , body, null, null, null);
                System.out.println(deploy);
        } catch (ApiException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

```
            }
        }

        public static void readDataFromSettings(Settings
            settings) {

            metadata.setName(settings.getName());
            selector.setMatchLabels(settings.getLabels());
            metadata1.setLabels(settings.getLabels());
            container.setImage(settings.getImage());
            port.setContainerPort(settings.getPort());
            container.setName(settings.getName());
            for (Map.Entry<String, String> entry : settings.
                getRequests().entrySet()) {
                    requests.put(entry.getKey(), new
                        Quantity(entry.getValue()));
            }
        }

        public static void writeDataToBody() {

            resources.setRequests(requests);
            ports.add(port);
            container.setPorts(ports);
            container.setResources(resources);
            containers.add(container);
            podSpec.setContainers(containers);
            podTemplateSpec.setMetadata(metadata1);
            podTemplateSpec.setSpec(podSpec);
            deploymentSpec.setStrategy(strategy);
            deploymentSpec.setSelector(selector);
            deploymentSpec.setTemplate(podTemplateSpec);
            body.setApiVersion("extensions/v1beta1");
            body.setKind("Deployment");
            body.setMetadata(metadata);
            body.setSpec(deploymentSpec);
            body.setStatus(status);
        }
}
```